An Automated Policy Refinement Process Supported by Expert Knowledge

Dissertationsschrift

in englischer Sprache
zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften (Dr. -Ing.)
vorgelegt von

Dipl.-Inform.
Taufiq Rochaeli
aus Bandung, Indonesien

Technische Universität Darmstadt
Fachbereich Informatik
Hochschulstraße 10
D-64289 Darmstadt

Erstreferent:	Prof. Dr. Claudia Eckert
Korreferent:	Prof. Dr. A Min Tjoa

Tag der Einreichung:	15.12.2008
Tag der Disputation:	12.03.2009

Hochschulkennziffer D17
Darmstadt 2009

Kurzfassung (Deutsch)

In einer durch Policy-gesteuerten Systemverwaltung spielt die Verfeinerung von Policies (Regelwerken) eine große Rolle. Der Verfeinerungsprozess transformiert die abstrakten Policies in umsetzbaren Policies, die durch den Policy-Enforcement-Point durchgesetzt werden. Dieser Prozess wird üblicherweise manuell durchgeführt.

Der manuelle Verfeinerungsprozess hat jedoch einige Nachteile. Einerseits benötigt dieser Prozess Expertenwissen, um die Policies zu verfeinern, andererseits, ist dieser Aufwand dieses Prozesses so aufwändig, dass einige Probleme, z.B. unpassende Spezifikation von umsetzbaren Policies auftreten können.

Um diese Nachteile zu beseitigen wird in dieser Dissertation der automatische Verfeinerungsprozess, der die Verfeinerungsmuster anwendet, vorgeschlagen. Durch die Adoption des Pattern Ansatzes werden Verfeinerungsmuster konstruiert, um Expertenwissen zu dokumentieren. Weiterhin werden die Verfeinerungsmuster, die Policies und das System formalisiert, so dass der automatisierte Verfeinerungsprozess realisiert werden kann.

In dieser Dissertation werden folgende Bausteine präsentiert: (i) Formalisierung des Systems und der Regelwerke (Policies), (ii) Definition des Verfeinerungsbaums der Regelwerke, (iii) formale Definition der Verfeinerungsmuster, (iv) Kombination von temporaler Aussagenlogik CTL* und Beschreibungslogik um die automatische Mustersuche zu realisieren und (v) Entwicklung eines Algorithmuses zur Verfeinerung der Regelwerke und zur Generierung von umsetzbaren Policies.

In dem Beispielszenario dieser Dissertation werden Workflows berücksichtigt, weil Regelwerke für Zugriffskontrolle generiert werden sollen. Diese Regelwerke werden aus Workflows abgeleitet. Diese Regelwerke werden üblicherweise in den Terminologien, die spezifisch für bestimmte Bereiche verwendet werden, ausgedrückt. Diese Bereiche sind beispielsweise der Finanzbereich, die Medizin und die Militär. Daher unterstützt diese Tatsache die Anwendung von der semantischen Technologie um das Expertenwissen zu formalisieren. Dennoch kann der in dieser Dissertation entwickelten Ansatz in anderen Sys-

temen, z.B. diskreten System, eingesetzt werden.

Der Beitrag dieser Dissertation ist ein Konzept zur Automatisierung des Policyverfeinerungsprozesses. Dazu wird das Sicherheitsexpertenwissen, das als Verfeinerungsmuster dokumentiert wird, angewendet. Um diese Konzept zu realisieren, Model Checking und Wissensrepräsentation wurden kombiniert.

Abstract (English)

In a policy-based system management, a policy refinement process is required to translate abstract policies, which are specified by human, into enforceable policies, which are enforced by machine.

However, a manual policy refinement process imposes some problems. The first problem is that it requires expert knowledge to perform the policy refinement process. The second problem is that refining policies for complex systems is a tedious task. Manual refinement process may cause some negative consequences due to human errors, i.e., improper specification of enforceable policies.

In order to solve the problems mentioned above, we envisage the automated policy refinement process by using refinement patterns. By adopting the pattern paradigm, we define the refinement patterns to capture the expert knowledge. Furthermore, we formalize these refinement patterns, the policies and the considered system. This approach enables the automation of the policy refinement process, which solves the second problem.

We present these building blocks: (i) formal representations of the considered system and the policies, (ii) definition of a policy refinement tree, (iii) definition of policy refinement patterns in formal representation, (iv) combining computational tree logic-* and description logics formalisms to enable the automated pattern matching and (v) development of the algorithm for policy refinement and for generating the enforceable policies.

In this thesis we consider the refinement of workflow policies as our scenario, since we want to specify access control policies. These access control policies are derived from workflow. These policies are usually specified in domain-specific terminologies, such as finance, engineering, military, etc. Thus, it underpins our approach to using semantic technology to formalize the policy refinement patterns. Although the work presented in this thesis deals with the refinement of workflow policies, one can adapt this approach to refine policies for various discrete systems.

The contribution of this thesis is a concept to automate the policy refinement process by using security experts knowledge, which is stored as policy

refinement patterns. To realize this concept, we combine two approaches, namely model checking and knowledge representation.

Acknowledgements

This research work could not have been realized without the help of other people. My first and greatest thanks goes to Prof. Dr. Claudia Eckert for giving me the opportunity to work in her research group and to conduct my research. I also appreciate her advice, her patience and her time in supporting my research.

I also thank Prof. Dr. A Min Tjoa for his willingness to take on the role of external supervisor.

Special thanks go to everyone in the research group "IT Security". They provided a joyful working atmosphere during my research work.

I also thank Ruben Wolf, Roland Rieke and Jan Peters, my colleagues in the Sicari project, for their support in this project. Additionally, I also want to thank the BMBF (Bundesministerium für Bildung und Forschung), who financed the Sicari project. I am also very grateful to Mrs. Angela Lloyd and Dr. Thomas Stibor for their help in the last phase of my thesis.

I also dedicate my special thanks to my parents Herlina Rochaeli and Achmad Rochaeli. Their support and encouragement has helped me to overcome the hardship during my research. Last, but not least, I also thank my wife Like Sisca for her support and patience.

Declaration

These doctoral studies were conducted under the supervision of Prof. Dr. Claudia Eckert. The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Department of Computer Science at Technische Universität Darmstadt as a candidate for the degree of Doktor der Ingenieurwissenschaften (Dr. - Ing.). This work has not been submitted for any other degree or award in any other university or educational establishment.

Darmstadt, December 15, 2008

Contents

Chapter 1

Introduction

The increasing complexity of IT-system demands flexibility in their management. Hence, it requires a mechanism that allows the administrator to alter the behavior of the managed system without modifying and compiling the source code of the software that controls the system during the runtime. In the past decades, some research work has been conducted [57, 92] that aim for a flexible system configuration. Such a system is also called a *policy-driven system*. The behavior of this system is controlled by the policies of the system. Thus, the system allows the alteration of its behavior during run-time only by pushing new policies into the system. Policy-based system management has gained more attention in the last decades. It enables the dynamic change of software behavior without too much effort, i.e., modifying and recompiling the source code.

1.1 Problem Statement

A policy refinement process addresses the gap between the security policies specified by the stakeholders (human) and the security policies that should be enforced by the machine. The security policies specified by humans are usually very general and abstract. For example, the scope of the target of the security policies is very general and wide. And also, the expressed desired property of the target is very general. On the other hand, the security policies that should be enforced by the machine are very concrete. The low-level security policies languages are only specific to each kind of policy enforcement engine, i.e., XACML, IPtables configuration, Apache directives, etc. In order to derive the enforceable policies, the abstract policies are refined manually. However, manual policy refinement process is error-prone due to the high complexity of the managed system.

Based on the latest research in policy refinement, we identify the following two major questions:

- The policy refinement process is still performed manually. How can we refine this automatically?
- In particular, the policy refinement process is performed by security experts, who have the necessary knowledge to refine the policy. How can we use expert knowledge to establish the automated policy refinement process?

Analogy of Policy Refinement Process. We present an analogy between the policy refinement process and the driving activity in order to better understand the concept of the policy refinement process. Let us consider the driving scenario shown in Figure 1.1.

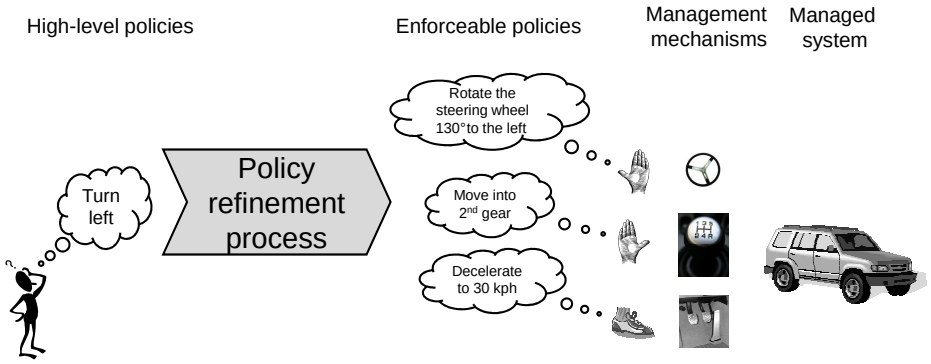

Figure 1.1: Policy refinement process: an analogy

This picture shows the scenario of a person driving a car, who wants to turn the car to the left. We assume that there are no obstacles standing in front of the car. In this scenario, the driver operates (or manages) the car (*managed system*) by manipulating the steering wheel, gear shaft, gas and brake pedals (*management mechanisms*). The intention of the driver, which is to turn the car to the left, is similar to the high-level policy of the stakeholders. In order to achieve this high-level policy, the driver has *to refine* the high-level policy *in his mind* into several low-level and enforceable policies, which can be *enforced* by the currently available management mechanisms. The low-level policies would be (i) turn the steering wheel to the left, (ii) move to the second gear, and (iii) decelerate to 30 kph.

As a proof of concept, we consider to refine workflow policies. In this case, the workflow is the considered system.

Sicari Project This thesis is undertaken within the framework of the Sicari (Sicherheitsarchitektur in ubiquitre Internetnutzung) project, which is funded by Federal Ministry of Education and Research. This project is supported by 18 academic and industrial partners. The goal of the Sicari project is to provide a platform consisting of security services. The security services range from the cryptographic modules, smart card-based authentication to the policy-based system management.

1.2 State-of-the-Art

Mont et al. presented a tool for policy refinement called POWER in [65]. Prior to using this tool, the domain experts should specify a set of policy templates, which are written in Prolog programming language. By using the keyword search from the Graphical User Interface of the tool, the user selects a policy template, which matches the security requirements. This template can be assumed as the abstract policy. Each policy template contains some terminologies of the information and system model, such as "employee", "file", "database". Furthermore, the POWER tool has also the database containing the hierarchies of the information and system model (ISM) terminologies. Thus, the POWER tool proposes then the refined policies of the policy template by substituting these abstract terminologies with more concrete terminologies with regard to the hierarchies defined in the ISM database. In the next step, machine-enforceable policies are generated from the refined policies. Therefore, this approach generates enforceable policies from the abstract policy. However, it is not clear whether this approach supports the hierarchical refinement process or not.

Bandara et al. propose a policy refinement process, which is based on the abductive reasoning technique [10]. Abductive-reasoning is a reasoning technique that derives the hypothesis a from the consequence b, such that $w \wedge a \Rightarrow b$ and $w \wedge a$ is consistent and w is an existing knowledge [14]. In their work, they assume that the abstract policies are the consequences and the refined policies are the hypotheses. As a formal foundation of this approach, they use Event Calculus to specify the policies [63]. Thus, a, b and w are event calculus formulas. For each refinement of an abstract policy, the policy administrator consults the refinement patterns, which are defined in the paper by Darimont and Lamsweerde [30]. Generally, the refined policies are still abstract and not concrete enough, so that a set of enforceable policies cannot be generated from these refined policies. Therefore, the refinement process is repeated until all abstract policies are refined into a set of concrete policies. Based on these policies, the policy administrator can generate a

set of machine-enforceable policies. In contrast to the approach of Mont et al., the policy refinement process should be performed manually by the user. Furthermore, the enforceable policies should also be generated manually from the concrete policies.

Rubio-Loyola et al. present an approach [82], which is based on the work by Bandara et al. In their work, they use the approach, which is similar to the approach of Bandara et al., to refine abstract policies into concrete policies. However, they use the automated model checking technique to automatically generate the machine-enforceable policies from concrete policies. In their work, the concrete policies are represented in temporal logic formulas. On the other hand, they specify the behavior of a managed system in a finite state machine. Thus, the model checking finds the execution traces, which should be controlled by the concrete policies and generates the enforceable with regard to the concrete policies.

Su et al. consider another aspect of the policy refinement. Their work deals with the refinement of the access control policies. They state that there are hierarchies of subjects and objects, which can be used to refine access control policies. Generally speaking, this approach is very similar to the approach of Mont et al. [65].

1.3 Solution

In order to solve the problems mentioned above, we envisage to automate the policy refinement process. As a proof of concept, we consider the refinement of workflow policies. To achieve this goal, we envisage the following milestones:

- Formal representations of the behavior of the system and the policies:
  We require the formal representation of the behavior of the system and the policies, since we want to perform an automated policy refinement process.

- Definition of policy refinement tree:
  Each refinement of an abstract policy generates a set of refined policies. These policies influence the fulfillment of the corresponding abstract policy. The abstract policy is fulfilled by fulfilling either all or some of its refined policies. Since these refined policies can be further refined, there exists a tree-like structure of the parent-children relationships between the policies. Generally, a policy refinement tree has an abstract policy as its root and a set of concrete policies as its leaves. Thus, the

fulfillment of the abstract policy at the root can be determined by the fulfillment of the policies in the leaves of the tree.

- Defining policy refinement patterns in formal representation adapted from pattern paradigm:
  This is one of the important building blocks that enable the automated policy refinement process. The aim of this building block is to be able to capture the expert knowledge of refining policies. Furthermore, this knowledge should be machine interpretable.

- Combining CTL* and DL formalisms to facilitate the automated pattern matching:
  This is the other important building block. It allows the automated pattern matching, without human supervision.

- Defining the algorithm for policy refinement and for generating the enforceable policies:
  The algorithm makes use of the previously mentioned building blocks in order to realize the automated policy refinement process.

1.4 Scientific Contribution

The policy-based system management gives more flexibility to control the managed system behavior. However, the increasing complexity of the managed system has increased the gap between human intention and the managed system behavior. The automated refinement process, which is based on expert knowledge, should reduce the gap between human intention and the managed system behavior. This should improve the management of the complex systems. In particular, the change in high-level policies can be immediately propagated to the managed system. Furthermore, the error-prone task in manual policy refinement process can be avoided.

Our approach to policy refinement uses the pattern paradigm, which aims at capturing the expert knowledge about refining policies. These patterns are applied in the policy refinement process. Thus, a novice administrator can refine the policies without having a deep knowledge about the managed system.

On the other hand, the current state-of-the-art of the pattern formalization approaches only allow us either to semi-automatically search the patterns or to automatically instantiate the patterns. Our approach allows us to fully automate the pattern matching and pattern-instantiation process. We achieve our goal by formalizing the considered system and formalizing the

description of the pattern. However, there are many domain specific terminologies used to describe the patterns, which have equivalence and generalization relationships between them. Additionally, we also use the description logic formalism in order to enable the semantic interpretation of these terminologies that improves the automated pattern search. Therefore, our study should give an insight in automating the pattern-based engineering process in any other fields.

Finally, we also present the combination of description logic with temporal logic, which enables to perform model checking based on the description logic reasoning engine. This method allows us to perform model checking, although the atomic propositions used in the model and in the formulas are from different domains provided that the ontologies bridging the domains are available. Although the idea of combining description logic and temporal logic is not new, several approaches exist only as concepts, because a new kind of reasoning engine should be built and its reasoning complexity is still too high [7, 89, 108]. On the contrary, our method does not require the construction of a new reasoning engine, since it can use the reasoning engine that supports the $\mathcal{SHK}$ description logic language and has the fixpoint semantics.

Make use of Workflow as the Considered System It should be noted that, as a proof of concept, this study makes use of workflow as the considered system. Therefore, the examples and the definitions presented are in favor of workflow. Nevertheless, one may also use the approach presented in this study to automate the policy refinement process for any other systems, for example, firewall router, web server, embedded systems, etc, as long as their behavior can be formalized as Kripke models.

1.5 Thesis Organization

The relationships between the chapters in this thesis are presented in Figure 1.5.

Chapter 2: Workflow and Policies. Since we consider the refinement of workflow policies as the proof of concept, this chapter initiates the discussion about the formalization of a workflow. Subsequently, a discussion of the need for policies, which is caused by the gap between the design and implementation, is presented. Finally, this chapter presents the method to formally specify the policies, in this case, the workflow policies.

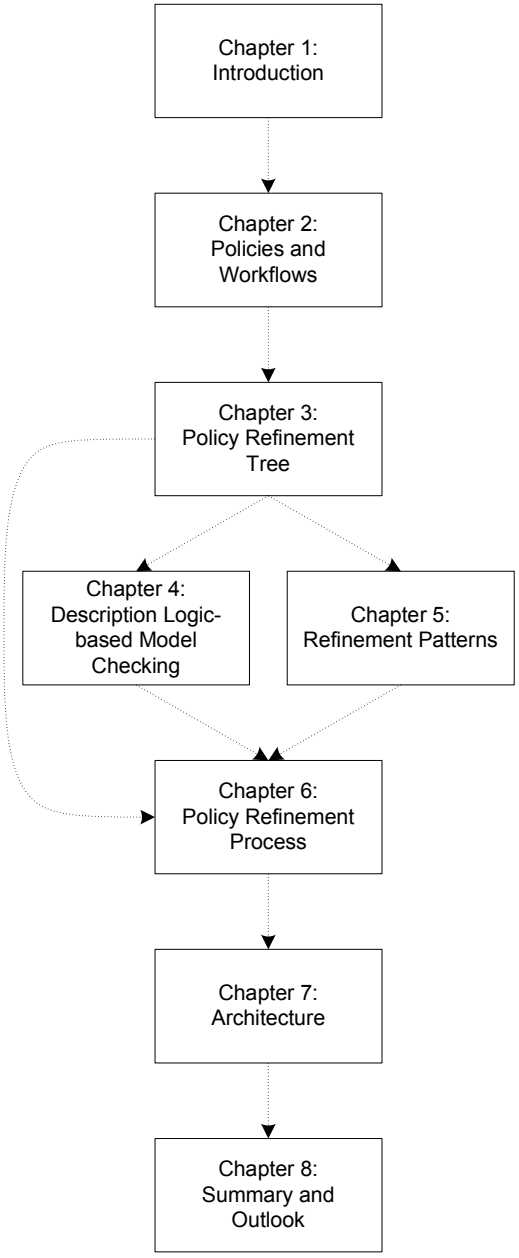

Figure 1.2: The chapters and their dependencies

Chapter 3: Policy Refinement Tree. This chapter defines the policy refinement tree, which should give the clear overview of the policies specified by the stakeholders and the policies generated by the tool. At the end of this chapter, we require that in order to automatically generate the policy refinement tree, we need to capture the expert knowledge in refining policy and to establish the formalism that supports the automated pattern matching.

Chapter 4: Description Logic-based Model Checking. We identify that pattern matching problem is similar to the model checking problem. Thus, in this chapter, we present the description logic-based model checking, which facilitates the automated pattern matching. We establish the description logic-based model checking by combining both formalisms, namely the description logic formalism and the temporal logic formalism.

Chapter 5: Refinement Patterns. Inspired by the pattern paradigm, which has been successfully adapted in the software engineering area, we want to follow the pattern paradigm to capture the expert knowledge in refining policies as refinement patterns. Furthermore, we also present the formalization of the refinement patterns based on the formalism, which is defined in the previous chapter.

Chapter 6: Policy Refinement Process. This chapter proposes the policy refinement algorithm by using existing building blocks, which are defined in Chapter 4, Chapter 5 and Chapter 6.

Chapter 7: Architecture. This chapter gives an overview of the implementation architecture of the tool and its module. It also explains the tools and standards used.

Chapter 8: Summary and Outlook. This chapter summarizes and presents the conclusion of our work and gives an outlook to the future work.

Chapter 2

Workflows and Policies

2.1 Introduction

The increasing complexity of a computer system and the flexible management of a computer system drive the adoption of policy-driven computer system management. This approach allows the behavior of the system to be managed without modifying and re-compiling the source code of the system's software. The behavior of a policy-driven system is controlled by some *policy enforcement points* and a *policy decision point* within the system. More specifically, each access request by subjects is intercepted by the policy enforcement point. To decide whether the request is denied or allowed, the policy enforcement point consults the policy decision point by forwarding the access request. In return, the policy decision point typically gives a *yes* or *no* response. In other words, the policy decision point has the capability to decide the request. To facilitate the decision process, the policy decision point typically consists of a reasoning engine and a policy database. Each request received by the policy decision point is immediately forwarded to the reasoning engine. Thus, the reasoning engine retrieves the policies, which are relevant to the considered access request.

The flexibility in the management of the complex system, using policy mechanisms, provides a significant advantage in computer system management. The architecture is shown in Figure 2.1.

At the same time, the adoption of information technology to support businesses is also increasing. The business processes of the firms or organizations are automated by IT systems. These automated business processes are known as workflows. Certainly, the execution of the workflows should be monitored and governed to avoid undesired execution of the workflows in order to meet stakeholders' objectives. These objectives can be achieved

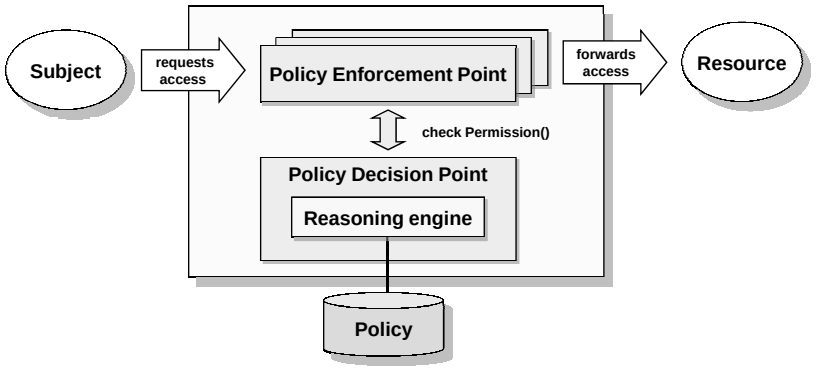

Figure 2.1: The common reference architecture of policy-based system management

by employing the policy-based system management approach. Further, the policy-based management of the workflows allows the immediate execution of the stakeholders' objectives.

Outline. This chapter gives a brief explanation of security policies, workflows and workflow security policies. The outline of this chapter is as follows:

Section 2.2 introduces the definitions of security policy from several sources. It also presents our definition of security policy, which is used within this thesis. Section 2.3 introduces the workflow and explains the kind of workflow, which is supported in this thesis. In this case, we consider only the workflow that can be specified by the UML activity diagram. Further, we also present the foundation of the workflow formalization. We use the Kripke model to formalize the workflow. In Section 2.4 we argue the necessity of the workflow policies. Additionally, we present the definition of the workflow policy in this section. Section 2.5 closes this chapter with a summary.

2.2 Security Policy

There are several definitions of security policy. Peltier defines a security policy as a statement of organizational goals or objectives, which aim for the protection of organizational assets [75]. Eckert states that a security policy describes the rules, users' behaviors and security measures, which are should be taken in order to achieve the security requirements of a considered system or an organizational unit [35].

On the other hand, we define a security policy as follows:

Definition 1 (Security Policy) *Let the managed system be a finite-state automaton. A security policy is a statement that contains the rules or security measures that should be applied to the managed system. Furthermore, the security policy partitions the states of the system into a set of secure states and a set of insecure states and allows the system only to run in secure states.*

In other words, the security policy governs the behavior of the system so that it avoids the states, which can lead to the unauthorized information modification or extraction. It should be noted that the definition of the security policy is adopted from the security policy definition in [15] and [35].

Wies suggested the security policy life-cycle in policy-driven system management, which is depicted in Figure 2.2.

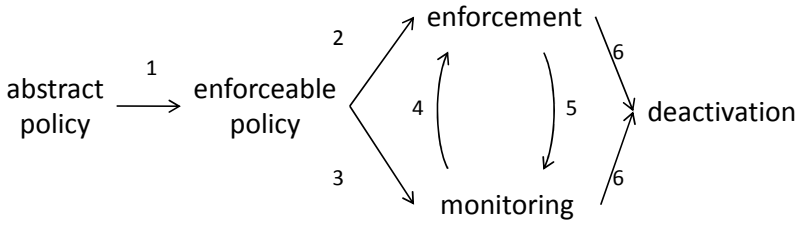

Figure 2.2: The life-cycle of security policies (taken from [106])

The policy life-cycle begins with the specification of the high-level policies. These policies are specified by humans and typically have an informal representation. The abstract policies are then refined into enforceable policies (step 1). Thus, the concrete policies can be interpreted by machines. Eventually, these concrete policies are also verified in order to check whether they conform to the security requirements or not. It should be noted that the concrete policies can either be enforced in the managed system or be applied as the monitoring policies. The enforcement policies first carry out certain actions and later monitor changes (step 2). The monitoring policies passively monitor certain managed sub systems and possibly react if necessary. It should be noted that a change in the management policies may also lead to a change in the enforcement policies (step 4). Consequently, a change in the enforcement policies may also lead to a change in the management policies (step 5).These changes may happen during the runtime process. Finally, the policies may become obsolete and be replaced with a new set of policies (step 6).

As we could see from the brief explanation of the policy life-cycle above, this thesis is concerned with the refinement process denoted by step 1 of Figure 2.2. Our goal is to refine abstract policies, which are specified by

human, into enforceable policies, which can be interpreted and enforced by machine.

2.3 Formal Representation of Workflow

Every organization has its own activities that aim to fulfill its organizational goals. These activities are represented as business processes. However, the rapid adoption of IT systems enables the automation of the business processes. The automated business processes are thus called workflows.

Several workflow representation languages and models exist, such as Business Process Execution Language (BPEL) [72], Business Process Modeling Notation (BPMN) from Object Management Group [46], XML Process Definition Language (XPDL) from Worfklow Management Coalition (WfMC) [110], and Petri Net [76], which have their own execution semantics.

Nevertheless, in this thesis we only consider the workflows, which are represented in UML activity diagrams [17]. Figure 2.3 shows an example of the workflow represented in UML activity diagram.

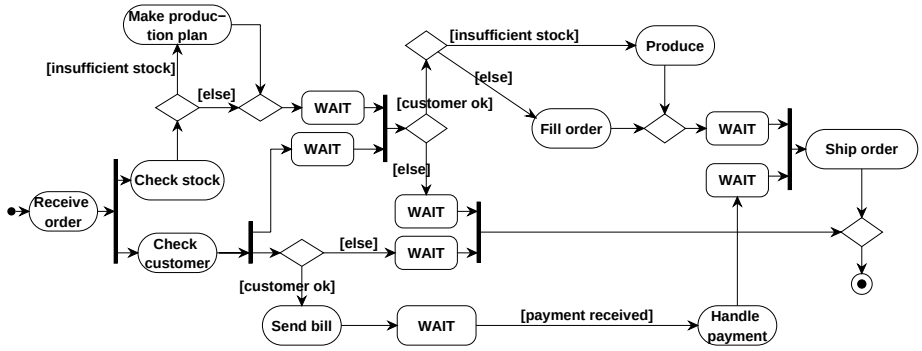

Figure 2.3: An example of Workflow represented in UML activity diagram (taken from [38])

Such workflows are presented by Eshuis et al. [38, 39] and Dumas et al.[34]. Furthermore, Eshuis et al. also presented the formalization of the activity diagrams into finite state machines [37], which can be translated into Kripke models [23]. Therefore, the Kripke model can be used to formalize the workflow. Theoretically, we can also use other notations and models specifying the workflows, if and only if, their execution semantics can be formalized using Kripke model. However, we do not consider workflows with multiple execution instances introduced in [100].

The Kripke model is named after its inventor, Saul Kripke, who invented this model to represent the "possible world semantics" [56]. A Kripke model has a set of states, which represent the "possible situations in the world", which may occur. Each state can be labeled by atomic propositions. The atomic propositions represent the *occurrence* of some certain events within this "snapshot" of the world. Additionally, the model also has a set of state transitions, which depict the possible "changes" from one situation into other situations. The Kripke model is formally defined as follows:

Definition 2 (Kripke Model) *A Kripke model K is defined as*

$$K : \langle W, I, L, RL, AP \rangle.$$

W is a set of states, $I \subseteq W$ is a set of initial states, $RL \subseteq W \times W$ is a set of state transitions, AP is the set of atomic propositions and $L : W \rightarrow 2^{AP}$ is a labeling function that maps a state to a subset of atomic propositions. The atomic propositions symbolize the property of the system that persists in each particular discrete state of the system.

An *execution trace*[1] of the Kripke model is a finite or infinite sequence of states starting from the initial states. It indicates the "history" of the state transitions of the considered system that could be interpreted as the behavior of the system. For example, an execution trace σ is described as a sequence $s_0 s_1 s_2 \ldots$. Hence, $\forall i \in \mathbb{N} : (s_i, s_{i+1}) \in RL$.

The definition of the Kripke model says that a state represents the occurrence of some certain events within this "snapshot" of the world, or more precisely, the system. Since we use the Kripke model to formalize the workflow according to Eshuis [37], the invocation of an external function is represented as an atomic proposition that labels the state of the Kripke model. However, the exact execution details of this function is outside the scope of the Kripke model.

The aim of the workflow formalization is to provide a foundation for the automated refinement process of workflow policies. In the proposed automated refinement process, which will be presented later, a set of execution traces in the workflow should be automatically matched. Therefore, the formalization of the workflow policies allows us to build an algorithm and to reuse an existing algorithm that automatically finds a set of execution traces.

Our choice of the formalization method, which is the Kripke model, based on the argument that we want to be able to characterize a class of execution traces as a formula. Obviously, finding a class of execution traces that match

[1] also known as path

a specific characteristic can also be done by employing the subgraph matching technique [25, 61]. However, specifying a class of execution traces by defining a graph is really impracticable, especially when specifying a complex class of execution traces. Since we use Kripke model, we can characterize a class of execution traces as a temporal logic formula.

2.4 Workflow Policies

As we can see in the previous section, the common workflow specification languages only specify the task execution sequences and do not specify any security constraints. Therefore, these workflow specification languages do not accommodate any security requirements [107]. This fact also implies that each state of the Kripke model only has the atomic propositions, which denote the execution of the tasks. However, in each state of the Kripke model, any other occurrence of significant events may also occur, for example, a malicious event.

Let us consider the sequence $\sigma = s_1 s_2 s_3$. The execution trace σ consists of a sequence of three states. According to Section 2.3, each state contains the execution of a certain task, which is denoted by the corresponding atomic proposition as shown in Figure 2.4. The trace σ denotes the execution of *create loan application, check loan application* and *approve loan application*, subsequently.

s_1 : create loan application
s_2 : check loan application
s_3 : approve loan application

Figure 2.4: The sequence of a loan application process

Suppose that a user performs the task sequence represented by σ by using role "manager". This activity is denoted as sequence σ_Φ, which states are shown in Figure 2.5.

Note that, in each state of the trace σ_Φ an additional proposition is introduced, namely the "user activates role manager". The atomic proposition indicates that in each state of the Kripke model, the user subsequently executes the corresponding tasks and also *activates* the role manager. The execution of such a sequence may be harmful to the bank, which runs the

s'_1 : create loan application; user activates role manager
s'_2 : check loan application; user activates role manager
s'_3 : approve loan application; user activates role manager

Figure 2.5: The sequence of a loan application process performed by a user

workflow, since a single user may create a fictitious loan application and approve it by himself [85].

This simple example shows the argument that the workflow constraints are still required in the workflow. Therefore, the security experts analyze the workflow and identify any potential security threats that may arise within the workflow. As countermeasures against the identified security threats, the security experts specify the workflow constraints to restrict the unwanted behavior of the workflow in a certain sub sequence of the workflow.

Matching to our needs, we define three classes of workflow policies, they are *specified workflow policy*, *abstract workflow policy* and *concrete workflow policy*. They are defined as follows:

Definition 3 (specified workflow policy) *Recall that W and AP are from Kripke model K. A specified workflow security policy is a tuple of $(\gamma, P) \in 2^W \times AP$, which specifies that property P should hold in each state $s \in \gamma$. The set of specified workflow policies is defined as N_{spec}.*

This definition is very practical for the security administrators, since it allows them to specify a workflow policy, which asserts that property P should hold in *every* state $s \in \gamma$, without explicitly specifying the order of the states.

For example, the specified workflow policy can be specified as $(W, secure\_application\_process)$. W contains all states of Kripke model K, and *secure_application_process* is the desired security property, which should hold in every state of the Kripke model.

On the other hand, the policy refinement process generates workflow policies, which explicitly denote the *sequence* of the states. First, we present the definition of the set of finite traces of Kripke model K.

Definition 4 (set of finite traces) *The set $Fragment(K)$ is a set of all possible finite traces in Kripke model K. It is defined as:*

$$\{s_0..s_n | \exists n = 0 : s_n \in W \vee \exists n > 0, \forall i \in \{1, .., n\} : s_i \in W \wedge (s_{i-1}, s_i) \in RL\}$$

where W and RL is the set of possible worlds and relations of Kripke model K, respectively.

Thus, we propose the definition of *abstract workflow policy* and *concrete workflow policy* as follows:

Definition 5 (abstract workflow policy) *An abstract workflow policy is a tuple $(\sigma, P) \in Fragment(K) \times AP$, which asserts that every state of sequence σ has the label P. σ is a fragment of task execution sequence in the workflow and P is a state label representing the desired workflow security property that should hold in every state of sequence σ. The set of abstract workflow policies is defined as N_{abs}.*

As an example for the abstract workflow policy, we present a policy $T = (\sigma, prevent\_fictive\_loan)$ that should prevent the fraud, which aims at creating a fictitious loan application in task sequence σ.

Definition 6 (concrete workflow policy) *A concrete workflow policy is a tuple $(\sigma, P) \in Fragment(K) \times AP$, which asserts that every state of sequence σ has the label P. σ is a fragment of task execution sequence in the workflow and P is a state label representing the application of a certain security mechanism or authorization constraint that should be applied within sequence σ. The set of concrete workflow policies is defined as N_{con}.*

An example for the concrete workflow policy according to Definition 6 is $(\sigma, separate\_the\_execution\_of\_first\_and\_last\_task)$. This policy states the application of authorization constraint within the sequence σ. The constraint restricts the execution of the first task (create loan application) and the last task (approve loan application) of sequence σ.

2.5 Summary

This chapter serves as a brief introduction to the security policy, workflow formalization and workflow policy.

In the beginning, we presented some definitions of security policy, which are not exactly the same, but have some commonalities between them. In a nutshell, the policy controls the behavior of the managed system. We also presented the definition of security policy. This should help the reader to get an overview of the security policy terminology. Subsequently, we presented the life-cycle of policies in order to understand the whole process involving policies and to identify the step in the life-cycle, in which we make our contribution to. This step is known as *refinement process*.

The aim of the security policy is to control the behavior of the managed system, such that it meets the security objectives of the stakeholders. Hence, the source of the security policies is the intention of the stakeholders to protect their asset, which is the managed system, and the target of the security policies is the managed system, on which the security policies should be enforced.

Since we deal with the refinement of the workflow policies, we should first explain workflow, before we can discuss workflow policy. Here,we stated one kind of workflow specification language, which is considered in our work. We currently consider the workflow specified in UML activity diagram, since it can be formalized by using the Kripke model based on the existing research work [37]. We consider using the Kripke model to formalize the workflow, because we can *characterize* a class of execution traces within the workflow by using a formula, in particular, a temporal logic formula. This is very useful, since we want to find a set of execution traces, in which a certain workflow policy applies.

Finally, this chapter provides a basis for the next chapter, which presents the policy refinement tree.

Chapter 3

Policy Refinement Tree

3.1 Introduction

In the previous chapter, we presented the approach to specify workflow policies of the stakeholders as state labels of the Kripke model and also some classes of policies. However, analyzing the fulfillment of the specified policies based on the application of the concrete policies is very difficult. Therefore, we need an hierarchical representation of the relationships between the policies.

Motivation. The term of policy hierarchy has been coined by Moffett et al. [64] and Maullo and Calo [21]. However, only Moffett et al. pointed out that the policy hierarchy is a result of the refinement of abstract policies into a set of concrete policies. This is due to the fact that each refinement step of an abstract policy generates a set of sub policies. Furthermore, they argued that the hierarchical relationships between policies is required to determine the satisfaction of abstract policy. In other words, the fulfillment of the abstract policy strongly depends on the fulfillment of its sub policies. Bandara et al. introduced the AND-branches (OR-branches) into the policy hierarchy [10]. An abstract policy, which is connected to its sub policies through AND-branch (OR-branch), is satisfied, if and only if, all (some) of its sub policies are satisfied. Therefore, the policy hierarchy should help us to analyze the fulfillment of the abstract policies.

The aim of this construction is very similar to Fault Tree Analysis [11, 105]. Fault Tree Analysis aims at finding the *minimal cut set*, which is a collection of basic event failures that can lead to the occurrence of the top-event [11]. On the other hands, policy refinement tree also aims at determining the set of concrete policies, which should be applied and preserve the fulfillment

of the higher policies. Usually, the administrators strive to fulfill the specified policies only by applying a minimal set of concrete policies. This is due to the fact that the application of concrete policies may induce additional cost or even constrain the activities of the users participating in this workflow.

To summarize, we need an hierarchical representation of the relationships between the policies, because we need to determine the fulfillment of the abstract policies that depend on the application of the concrete policies.

Outline. Section 3.2 first presents our argumentation, on which our decision to consider the goal-oriented requirements engineering methods to construct the policy refinement tree is based. It then subsequently continues with the related work, which are the goal-oriented requirements engineering methods.

Based on the definitions of workflow policies presented in Section 2.4, we propose the construction of the policy refinement tree in Section 3.3. Section 3.4 defines the fulfillment of a policy refinement tree. Thus, Section 3.5 concludes this chapter.

3.2 Related Work

This section discusses the methods that are related to the policy refinement tree, which will be presented later. We divide this section into two parts. The first part discusses the Fault Tree Analysis method, which inspires the construction of the policy refinement tree. In the second part, we discusses some approaches that deal with the hierarchical representation of policies or goals.

3.2.1 Fault Tree Analysis

The concept of policy refinement tree is inspired by the Fault Tree Analysis method [11, 105]. The Fault Tree Analysis method is one of the failure analysis methods in field of engineering, especially, safety engineering field. This method was first developed in early 1960 to analyze the failures of failure-critical systems, i.e., nuclear plants. Thus, the Fault Tree Analysis method constructs a tree that depicts the event of failures.

The root node of the tree represents the "top-event of failure" of the considered (sub-)system. Starting from this node, further events, which influence the occurrence of the top-event, should be added.

In general, there are two categories of event defined in the fault tree analysis, they are *intermediate event* and *basic event*. The intermediate event

results from a combination of sub-events, which are connected through a logic gate. The "top-event" falls into this category. These gates are logical AND- and OR-gates. The events are further decomposed until, eventually, all intermediate events are decomposed into basic events. A basic event is the most primitive event, which is cannot be decomposed. Thus, the root and the inner nodes of the tree are intermediate events and the leaf nodes are basic events.

The fulfillment of the intermediate events, especially top-event, is determined by the fulfillment of their sub events, because each intermediate events is connected through the boolean gates. Furthermore, the fulfillment of an intermediate event can be expressed as a logical formula, which consists of a conjunction or disjunction of several fulfillments of the sub events. If the sub events are still intermediate events, the fulfillments of these sub events can be further substituted into the fulfillments of the basic events according to the structure of the Fault Tree.

In Figure 3.1, we can see the example of an Fault Tree model. The Fault Tree has the event called "Boiler level above limit when operating" as the top event. The "+" and "·" gates in Figure 3.1 represents the "OR" and "AND" gates, respectively. The fulfillment of the top-event can be expressed as the following formula:

$$\begin{aligned}(((&\text{sensor 1 failure } \wedge \text{ sensor 2 failure}) \vee \text{ control stuck on }) \\ &\vee \text{ pump stuck on}) \vee \text{level} > \text{limit upon startup.}\end{aligned}$$

The main goal of the Fault Tree Analysis is to find a *minimal cut set*, which is defined as a smallest combination of basic event failures, which, if they all occur, will cause the top event to occur [105].

3.2.2 Hierarchical Representation of Policies or Goals

Works by Bandara and Rubio-Loyola [10, 82] about policy refinement present the adoption of goal refinement tree [102, 101], which is introduced by Lamsweerde et al., to represent the policy hierarchy. In this sense, they share a common point of view, which states that a *goal* is almost similar to a *policy*. Furthermore, Maullo and Calo [21] define a goal as one of the abstraction levels in the policy hierarchy.

To have a common understanding of a goal, we present the definition of goal according to Lamsweerde et al. [102]:

Definition 7 *A goal denotes the desired objective that a considered system should meet.*

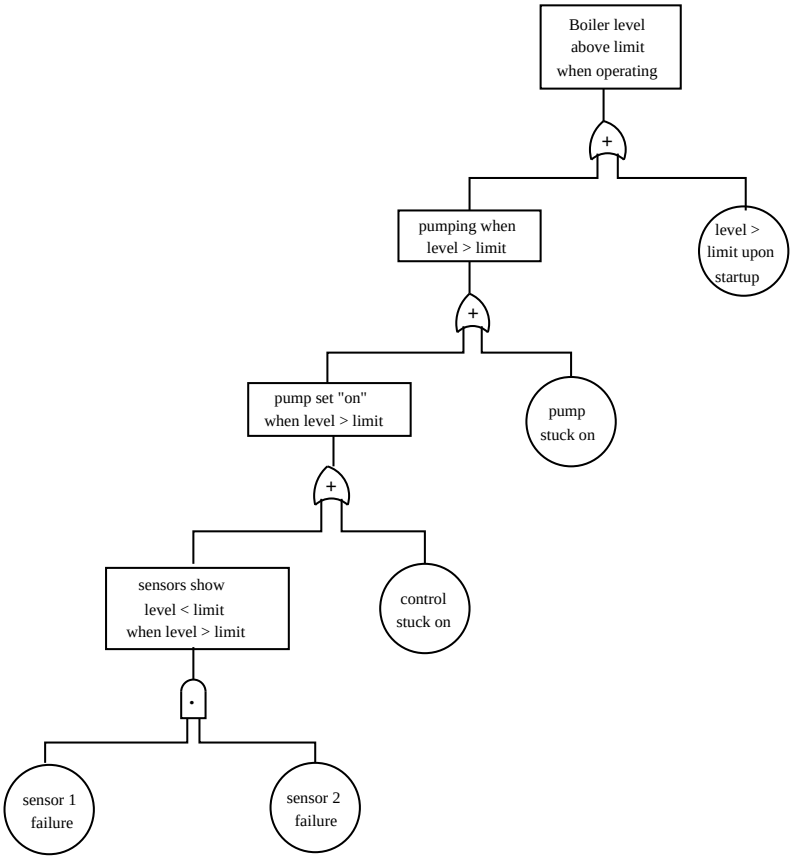

Figure 3.1: A Fault Tree for the Boiler System taken from [9]

Generally speaking, a goal is a statement that controls the behavior of the considered system in order to achieve the desired behavior. Intuitively, the purpose of a goal is very similar to the purpose of a policy, namely, to control the behavior of the considered system. Although these two terminologies are not the same, but their purpose is the same.

However, there are also another view that points out the differences between goal and policy. Feather pointed out in [40] that a policy is used to specify a desired goal. It means that the goal is more general and abstract than the policy and the policy is used to describe the goal in more detail. Sutcliffe et al. states in their paper that a policy is a class of goals [97]. Thus, they share the opinion that the goal is more general than the policy.

Nevertheless, both goals and policies share the same purpose, namely to control the behavior of the considered system. Therefore, we follow the approach of Bandara and Rubio-Loyola of adopting the goal refinement tree,

which originates from the goal-oriented requirements engineering, to represent the policy hierarchy.

The related work discussed below mainly deals with the goal-oriented requirements engineering. That means, these approaches make use of the concept "goal" to represent a requirement. The goal-oriented requirements engineering approach has recently been gaining more attention. The advantages offered by the goal-oriented requirements engineering are as follows (taken from [102]):

- A goal is capable of capturing the objectives, which the system under consideration should achieve, at different levels of abstraction.
- By specifying the requirements as goals, one could define the *influence* relationships between goals that specify whether the achievement of a goal *positively or negatively influences* the achievements of other goals. This enables the analysis of the dependencies between the requirements.
- Naturally, the goals at the higher level of abstraction are subject to the refinement process resulting in a goal refinement tree. The nature of the goal refinement tree provides a mechanism for structuring complex requirements documents. Thus, it also increases the readability of the complex requirements for the stakeholders.
- Moreover, the goal refinement tree provides the traceability links from high-level strategic objectives to the low-level technical requirements. This will greatly help the stakeholders to focus more on the management objectives than the technical details, but also allows the stakeholders to take a closer look at the low-level technical requirements fulfilling the high-level strategic objectives, if needed.

Obviously, the goal-oriented requirements engineering is not without any disadvantage. It still, however, lacks tool support and is not yet established as the use cases diagram of the UML standard.

KAOS

The KAOS approach has been developed by Darimont and van Lamsweerde et al. [101, 103, 102, 28, 29] as a framework to support the requirements engineering process by using the goal concept.

The KAOS approach captures requirements as goals, which have dual representations. The first representation is a text, which informally describes the goal. This representation provides the non-technical persons, such as

stakeholders, with an easily understandable formulation of the goal. The second representation, which corresponds to the first representation, is the formulation of the goal in terms of Linear-time Temporal Logic formalism. This formal representation has the purpose of enabling the precise formulation of the goals. Furthermore, this representation also allows the formal verification of the requirements and the generation of, i.e., code fragments and test cases [101].

This approach defines four classes of goals, they are *Maintain, Avoid, Achieve* and *Cease*. The goals of the types of *Maintain* and *Avoid* indicate that the goals should always be fulfilled or should never be fulfilled in the future, respectively. The goals of the types *Achieve* and *Cease* indicate that the goals should eventually be fulfilled in future, or should cease to be fulfilled in the future, respectively.

The interesting fact is that this approach is also used to elicit security requirements, which is presented in [104]. The security requirements are represented as a goal refinement tree. The elicitation of the security requirements is accomplished by constructing the *anti-model* tree. The anti-model tree represents the malicious intention of the attackers, which follows the idea of the attack tree presented by Schneier [90]. Nevertheless, the anti-model tree is more sophisticated compared to the original attack tree by formally representing the malicious intention. By considering the anti-model tree, one can concurrently derive the security requirements in the form of the goal refinement tree.

In Figure 3.2, we present an example of a goal refinement tree, which is taken from [28]. Each goal is represented by the rectangle node. It should be noted that there are two types of goals, ordinary goals and operationalizable goals. The ordinary goal resembles the abstract policy, which cannot be enforced and thus should be refined. On the other hand, the operationalizable goal, which is represented as a rectangle with bold line, resembles the concrete policy, which can be interpreted and thus enforced by the machine. It should be noted that each goal has either an AND-joint connection or an OR-joint connection with its sub-goals. These connections denote the fulfillment of the goals.

I-star

A similar framework that follows the KAOS approach is the I-star framework (I-*) introduced by Yu [111]. The I-* framework makes use of conceptual modeling to capture the requirements as goals. An actor, which can either be a human or an active software entity, plays a central role in this model. Every actor considered relevant in the requirement process is depicted as a

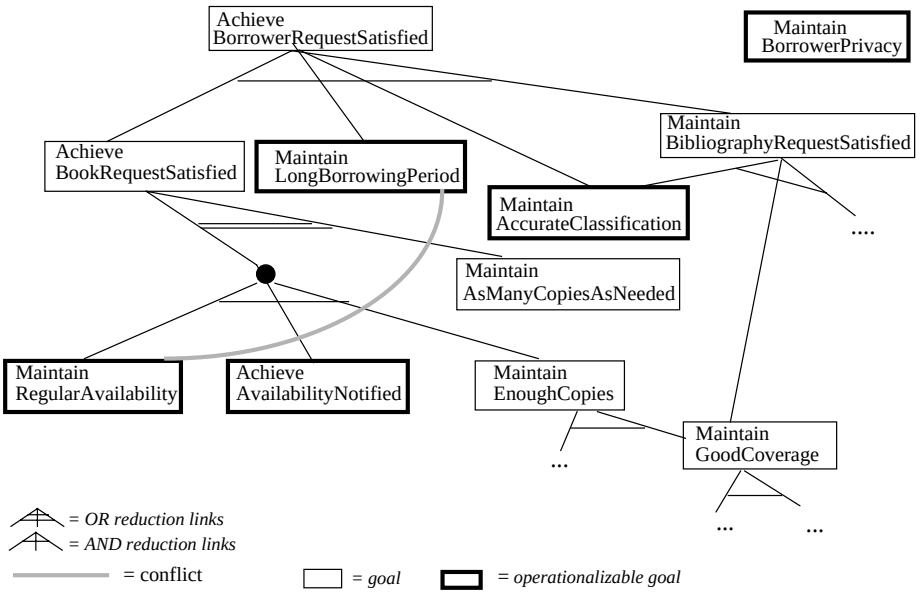

Figure 3.2: An example of goal refinement tree taken from [28]

node. These actors depend on each other and the dependency relationships between them are depicted as directed edges between the nodes. Each dependency relationship always has a *depender* and a *dependee*. These dependency relationships are classified into four categories: *resource dependency, task dependency, goal dependency* and *soft-goal dependency*. Nevertheless, our interest lies in the goal and soft-goal dependency relationships.

Both goal and soft-goal dependency relationships denotes that an actor (depender) depends on another actor (dependee) to make a certain goal true. Since the goal expresses only the desired properties or states of the considered system, the dependee has freedom to choose how it achieves the goal. The particular type of goal dependency, namely the soft-goal dependency, represents the requirement that can not be directly achieved by means of any implementation. Such a requirement is also known as a non-functional requirement and typically deals with the quality of a particular property of the system. With this fact, we learn that the goal is always related to the depender, who claims the achievement of the requirement, and to the dependee, who guarantees the achievement of this goal.

The usage of the conceptual modeling method in this approach gives an easy understanding to the stakeholders, although it lacks in the formal representation of the requirements.

AGORA

AGORA is an abbreviation of attributed goal-oriented requirements analysis, which is presented by Kaiya et al. [54]. Basically, it is an extension of the previously mentioned goal-oriented requirements engineering methods by introducing attributes to the goal refinement tree. They found out that the existing goal-oriented requirements engineering does not support some features, such as prioritizing and solving the conflicting goals, analyzing the impacts when the requirements change, etc. These shortcomings are addressed by adding contribution value and preference matrix to each edge and each node, respectively. The contribution value, which is represented as integer value, represents the degree of a sub-goal to the fulfillment of its parent goal, while the preference matrix represents the preference of the goal for stakeholders, for example, customer and developer.

3.2.3 Risk Assessment Using SemanticLIFE

Mansoor Ahmed et al. presented in [2] an approach to risk assessment in collaborative environments, which makes use of semantic web technology. The risk assessment process is realized by using the SemanticLIFE platform [3]. The extensibility of the platform is achieved through its modular architecture. The platform acquires the data containing the user annotations through some input plug-ins, such as Google Desktop2 captured data, communication logs, etc. The information about managed or analyzed objects are also passed by the corresponding plug-in. Both data objects are passed on to the analysis plug-in, which can perform the feature extraction methods and indexing techniques. This information is stored as meta-store, which schema is structurally defined as ontologies. The pre-defined core ontologies, which are used in the risk assessment process, are already defined in the repository.

Through the appropriate plug-ins one can define and modify the semantic relationships between the objects and more importantly, construct the semantic relationships to build the risk assessment model. As its proof of concept, the SemanticLIFE has been used to calculate the Annual Loss Expectancy.

Surely this approach would be such a great extension to our policy refinement tree model, since it incorporates a set of useful core ontologies.

3.3 Policy Refinement Tree

The related work discussed in the previous section influences our proposal for the policy refinement tree. Basically, the policy refinement tree is similar to the original Fault Tree Analysis. However, we use the set of previously defined policies as the nodes of the tree and present the formal semantics of the policy refinement tree. In the following we will define the structure of the policy refinement tree and then its formal semantics.

3.3.1 Structure of the Policy Refinement Tree

A policy refinement tree consists of policies representing specified (N_{spec}), abstract (N_{abs}) and concrete (N_{con}) policies. These policies are connected together through *gates*. More precisely, the "out-pin" of a gate is connected to a policy and the "in-pins" of the gate are connected to sub policies. By using the gates to connect the policies, we will expect a hierarchical structure of the tree. In the following, we present the definitions of the policies and the gates.

Definition 8 (policies of the tree) *Recall the definitions of N_{spec}, N_{abs} and N_{con} from Definition 3, 5 and 6. $N = N_{spec} \cup N_{abs} \cup N_{con} \cup \{\omega\}$ is the set of policies, whereas ω is the root policy.*

Definition 9 (*AND*-gates) *$AndGate \subset N \times 2^N$ is the set of* AND-*gates. Let $x, y_1, \ldots, y_n \in N$. If the fulfillment of x is equivalent to the logical conjunction of the fulfillments of $y_1, \ldots, y_n$, then an* AND-*gate is specified. x is the* output *of the gate and $y_1, \ldots, y_n$ are the* inputs *of the gate.*

Definition 10 (*OR*-gates) *$OrGate \subset N \times 2^N$ is the set of* OR-*gates. Let $x, y_1, \ldots, y_n \in N$. If the fulfillment of x is equivalent to the logical disjunction of the fulfillments of $y_1, \ldots, y_n$, then an* OR-*gate is specified. x is the* output *of the gate and $y_1, \ldots, y_n$ are the* inputs *of the gate.*

Definition 8 specifies the set of policies N, which consists of previously defined policies (specified, abstract and concrete policies) and a root policy ω.

Definition 9 and Definition 10 define the gates that connect a policy with its sub policies. Thus, the gates determine the decompositions of the policies. However, the semantics of these gates will be defined in the next section.

Example 1 *Let N_{spec}, N_{abs} and N_{con} be defined as follows:*

$$\begin{aligned} N_{spec} &= \{(\{s_1, s_2, s_3, s_4\}, secure\_loan\_appl\_process)\} \\ N_{abs} &= \{(s_1s_2s_3, prevent\_fictive\_appl), (s_1s_2s_3s_4, prevent\_unauth\_access)\} \\ N_{con} &= \{(s_1s_2s_3, separate\_first\_and\_last\_tasks), \\ &\quad (s_1s_2s_3, access\_only\_for\_role\_employee)\} \end{aligned}$$

For the sake of brevity, we substitute the policies with symbols:

$$\begin{aligned} P_1 &\stackrel{def}{=} (s_1s_2s_3s_4, secure\_loan\_appl\_process) \\ P_2 &\stackrel{def}{=} (s_1s_2s_3, prevent\_fictive\_appl) \\ P_3 &\stackrel{def}{=} (s_1s_2s_3, prevent\_unauth\_access) \\ P_4 &\stackrel{def}{=} (s_1s_2s_3, separate\_first\_and\_last\_tasks) \\ P_5 &\stackrel{def}{=} (s_1s_2s_3, access\_only\_for\_role\_employee) \end{aligned}$$

Then, the gates are specified as follows:

$$\begin{aligned} AndGate &= \{(\omega, \{P_1\}), (P_1, \{P_2, P_3\}), (P_2, \{P_4\}), (P_3, \{P_5\})\} \\ OrGate &= \emptyset \end{aligned}$$

Furthermore, we impose some restrictions to the gates. Before we introduce the restriction, we define the following relation:

Definition 11 (sub policies relation) *Let $d, e \in N$. We define the transitive relation $\stackrel{sub}{\rightharpoonup}$ as*

$$d \stackrel{sub}{\rightharpoonup} e \stackrel{def}{=} \exists (u, V) \in AndGate \cup OrGate : d = u \wedge e \in V.$$

Informally, this definition specifies the sub policy relation between a policy and its sub policy. Thus, $d \stackrel{sub}{\rightharpoonup} e$ says that e is one of the sub policies of d.

Example 2 *Recall the set AndGate from Example 1. Thus we have that $\omega \stackrel{sub}{\rightharpoonup} P_1$ and also $\omega \stackrel{sub}{\rightharpoonup} P_5$.*

After defining the sub policies relation $\stackrel{sub}{\rightharpoonup}$, we can define the relationships between the policies from different classes.

Definition 12 (sub policies of the root policy) *ω can only have specified policies N_{spec} as its sub policies.*

Example 3 *Let us consider Example 1. The sub policy of* ω *is* P_1*, which is an element of* N_{spec}*.*

Definition 13 (sub policies of specified policies) *Each policy of specified policies* N_{spec} *can only have either abstract or concrete policies (*N_{abs} *or* N_{con}*) as its sub policies.*

Example 4 *Let us consider Example 1. The sub policies of* P_1 *are* P_2 *and* P_3*, which are elements of* N_{abs}*.*

Definition 14 (sub policies of abstract policies) *Each policy of abstract policies* N_{abs} *can only have either abstract or concrete policies (*N_{abs} *or* N_{con}*) as its sub policies.*

Example 5 *Let us consider Example 1. The sub policy of* P_2 *is* P_4*, which is an element of* N_{con}*.*

Definition 12 - Definition 13 define the sub policies relationships between the classes of policies. These relationships can be depicted in Figure 3.3

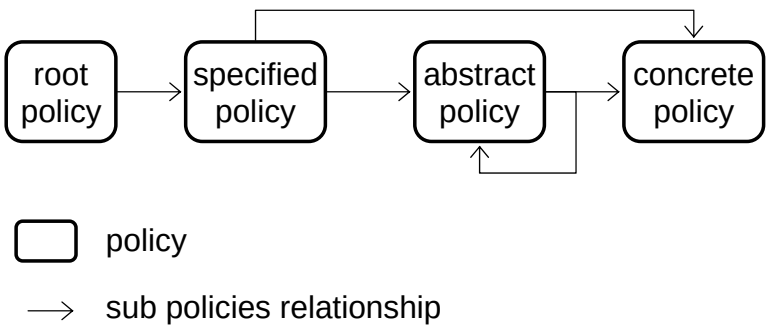

Figure 3.3: relationships between the classes of policies

Restriction to the gates The following restriction rules apply to the gates:

- if $(a, b) \in AndGate \cup OrGate \wedge (a, c) \in AndGate \cup OrGate$, then $b = c$ This restriction ensures the functional mapping relationship from each policy to a set of policies. It also implies that each policy can only have a single set of sub policies, which is connected through a gate.

- $\nexists x_n \in N : x_1 \overset{sub}{\rightharpoonup} x_2 \overset{sub}{\rightharpoonup} \ldots \overset{sub}{\rightharpoonup} x_n \overset{sub}{\rightharpoonup} x_1$
  This restriction forbids the cyclic sub policies relationships between the policies.

- $\forall x \in N : \omega \overset{sub}{\rightharpoonup} x \rightarrow x \in N_{spec}$
  This restriction says that all sub policies of the root policy ω are specified policies.

- $\forall x \in N_{spec}, \forall y \in N : x \overset{sub}{\rightharpoonup} y \rightarrow y \in N_{abs} \vee y \in N_{con}$
  This restriction says that all sub policies of each specified policy are either abstract policies or concrete policies.

- $\forall x \in N_{abs}, \forall y \in N : x \overset{sub}{\rightharpoonup} y \rightarrow (y \in N_{abs} \wedge y \in N_{con})$
  This restriction says that all sub policies of each abstract policy are either abstract policies or concrete policies.

3.3.2 Formal Semantics of the Policy Refinement Tree

Previously, we defined the structure of the policy refinement tree without any semantics. Therefore, we will present some definitions to build the formal semantics of the policy refinement tree.

Definition 15 (signature) *Since N is finite set, then N is also a countable set. Thus, we define N as the* signature.

Definition 16 (symbols) *We define the following symbols:*

- *T represents "true",*
- *F represents "false",*
- *$\otimes$ represents the conjunction symbol,*
- *$\oplus$ represents the disjunction symbol.*

Definition 17 (formula) *$Form$ is the set of formulas over Signature N. $Form$ is inductively defined as:*

1. $T \in Form$
   $F \in Form$
   $N \subseteq Form$
2. *if A, $B \in Form$, then*
   $A \otimes B \in Form$
   $A \oplus B \in Form$.

Definition 15 - Definition 17 construct the building blocks for the formulas over N. Example 6 shows us the formulas, which are constructed by these definitions.

Example 6 *Recall the set of policies* $\{P_1, \ldots, P_5\}$ *from Example 1. Thus,* $P_2 \otimes P_3$, $P_3 \oplus P_4 \oplus P_5$ *and* $P_1 \otimes P_2 \otimes P_3 \otimes P_4 \otimes P_5$ *are the elements of* $Form$.

Definition 18 (substitution) $subst : N \rightarrow Form$ *substitutes a policy, which is decomposed into several sub policies, into a conjunction or disjunction of its sub policies. It is defined as:*

$$subst(n) \stackrel{def}{=} \begin{cases} subst(a_1) \otimes \ldots \otimes subst(a_n),\ \exists(n, \{a_1, \ldots, a_n\}) \in AndGate \\ subst(a_1) \oplus \ldots \oplus subst(a_n),\ \exists(n, \{a_1, \ldots, a_n\}) \in OrGate \\ n,\ otherwise. \end{cases}$$

Example 7 *Recall the set AndGate and OrGate from Example 1. Then we have:*

$$\begin{aligned} subst(\omega) &= subst(P_1) \\ &= subst(P_2) \otimes subst(P_3) \\ &= subst(P_4) \otimes subst(P_5) \\ &= P_4 \otimes P_5 \end{aligned}$$

In the previous example we see that the *subst* function substitutes each policy with a conjunction of its sub policies. This function is recursively applied to the policies until the policy does not have any sub policies. This function is used when we want to evaluate the fulfillment of the policies.

Now we define the functions that are used to evaluate the fulfillment of the policies.

Definition 19 (policy application function) $app : N \rightarrow \{true, false\}$ *determines the application of concrete policy.*

$$app(n) \stackrel{def}{=} \begin{cases} true,\ if\ policy\ n\ is\ applied, \\ false,\ if\ policy\ n\ is\ not\ applied. \end{cases}$$

Example 8 *Suppose that all concrete policies from Example 1 are enforced. Thus, we define app as* $\{P_4 \mapsto true, P_5 \mapsto true\}$.

Definition 20 (evaluation function) *$eval : Form \rightarrow \{true, false\}$ evaluates each formula with the following rules:*

$$eval(T) = true$$
$$eval(F) = false$$
$$eval(N_{con}) = app(N_{con})$$
$$eval(a_1 \otimes \ldots \otimes a_n) = \bigwedge_{i \in \{1,\ldots,n\}} eval(a_i)$$
$$eval(a_1 \oplus \ldots \oplus a_n) = \bigvee_{i \in \{1,\ldots,n\}} eval(a_i)$$

Example 9 *let us consider the assignment of app function from Example 8. If we want to evaluate $P_4 \otimes P_5$, then*

$eval(P_4 \otimes P_5) = eval(P_4) \wedge eval(P_5) = app(P_4) \wedge app(P_5) = true \wedge true = true.$

Now we can define the policy refinement tree.

Definition 21 (policy refinement tree) *A policy refinement tree T_{pol} is defined as:*

$$T_{pol} : \langle \omega, N, AndGate, OrGate, app \rangle.$$

Example 10 *Recall N, $AndGate$ and $OrGate$ from Example 1 and app function from Example 8. T_{pol} is a policy refinement tree with:*

$$\begin{aligned} N &= \{\omega, P_1, P_2, P_3, P_4, P_5\} \\ AndGate &= \{(\omega, \{P_1\}), (P_1, \{P_2, P_3\}), (P_2, \{P_4\}), (P_3, \{P_5\})\} \\ OrGate &= \emptyset \\ app &= \{P_4 \mapsto true, P_5 \mapsto true\} \end{aligned}$$

The fulfillment of the policy refinement tree is defined as follows.

Definition 22 (fulfillment of the policy refinement tree) *The policy refinement tree is fulfilled if and only if, $eval(subst(\omega)) = true$.*

Example 11 *Consider the policy refinement tree T_{pol} from Example 11. We want to determine the value of $eval(subst(\omega))$.*

$$\begin{aligned} eval(subst(\omega)) &\overset{Example7}{=} eval(P_4 \otimes P_5) \\ &\overset{Example9}{=} true \end{aligned}$$

Since $eval(subst(\omega)) = true$, then the policy refinement tree is fulfilled.

3.4 The Fulfillment of the Policy Refinement Tree

The policy refinement tree is very similar to the Fault Tree Analysis (FTA) tree [105], which is widely used in industrial and mechanical engineering. Therefore, the fulfillment of the root policy of a policy refinement tree is also similar to the root event of a FTA tree. The fulfillment of the root policy depends on the application of the concrete policies denoted by the leaves of the tree.

Intuitively, one could fulfill the root policy by applying all concrete policies. But this assumption only holds if there is no conflict between the policies represented in the refinement tree. On the other hand, applying all concrete policies seems impractical. As a matter of fact, the presence of conflicts between the policies prevents the application of all concrete policies. Therefore, the security experts and administrators strive to find the minimal set of concrete policies, which can be applied to satisfy the root policy. Clearly, the set of concrete policies should be free from any conflicts. This problem is also known in the FTA research community as *minimal cut-set*. However, we consider this problem as beyond of this thesis.

3.5 Summary

We know that the common practices in representing security policies use different representations for different abstraction levels. However, this thesis tries to answer the following question: *"Is it possible, to perform an automatic refinement of the security policies?"* In our attempt to answer this question, we investigate the research work in the requirements engineering fields and adopt some ideas from this work. The most obvious adoption is the refinement tree that allows a uniform representation of security policies across different abstraction levels.

In this chapter, we present the definition of the refinement tree of the workflow security policy. Each node of this tree represents a security policy, which expresses the desired property within a fragment of an execution path of the workflow design. It specifies that all states within this fragment of execution path are labeled with the atomic proposition representing the desired behavior. It should be noted that the existence of the policy refinement tree always depends on the Kripke model representing the behavior of the workflow design. As a matter of fact, a policy refinement tree shows the *hierarchical structure* of the state labels representing the desired properties within the Kripke model.

We also present an example of the refinement tree of a security policy pertaining to the loan application process. As we can see from the example, the root security policy refers to a very broad fragment of the workflow execution path and also specifies a very general desired property. On the other hand, the security policies represented by the leaves of the tree refer only to small fragments of the workflow execution path and specify concrete workflow policies. This is made possible by the refinement process, which is performed by the security experts. This process analyzes the abstract security policies and specifies the sub-policies by attaching sub-nodes to the corresponding node.

We shall now raise two questions in regard to this refinement problem: (i) how can we automate the refinement process, and (ii) how can we capture the expert knowledge in refining the abstract security policies?

To continue our quest in answering these questions, the next two chapters present the methods we have developed, which enable the automated construction of the refinement process and facilitate the acquisition of security experts' knowledge in refining the workflow security policies.

Chapter 4

Description Logic-based Model Checking

4.1 Introduction

The policy refinement process constructs the security policy refinement tree introduced in Chapter 3 by attaching new nodes on to the lowest nodes of the tree. The new nodes represent the less abstract sub policies and the lowest nodes represent the unrefined abstract policies. Simultaneously, attaching new sub policies also defines some new state labels in the particular states of the Kripke model, which represents the workflow's behavior and the abstract workflow policies. These sub policies are proposed by the expert knowledge, which is documented as refinement patterns.

Thus, the policy refinement process can be generally seen as an iteration, which consists of two steps. They are the *pattern matching* and the *pattern instantiation* steps. The pattern matching step checks whether some particular refinement patterns can be used to refine the abstract policies. The pattern instantiation step adds new sub policies in the policy refinement tree and also adds some state labels into some particular states of the Kripke model. In this chapter, however, we will focus more on the pattern matching step.

The pattern matching step takes a refinement pattern and the Kripke model as its input and generates a set of fragments of the execution paths, which match the description of the abstract policies described in the refinement pattern. If the pattern matching step generates only an empty set, then the refinement pattern cannot be used to refine the currently available abstract policies.

Since we want to automate the pattern refinement process, we also want

to automate the pattern matching step. There are several approaches that can be used to perform the automated pattern matching, namely: (i) subgraph isomorphism matching, (ii) intersection between two automata and (iii) temporal logic model checking. The first and the second approaches only allow us to specify the pattern in the form of a graph or of an automaton, respectively.

We opted for the latter, namely the temporal logic model checking because it allows us to specify the pattern as a set of temporal logic formulas, which is more convenient to the pattern authors. The concept presented in this chapter has already been presented in [80].

Motivation. Our goal is to automate the pattern matching by using temporal logic model checking. The model checking technique is widely used in the software verification process. The purpose of model checking technique is to verify whether the implemented computer program (the model) satisfies the specified requirements (the formulas). The model checking technique has three basic steps, they are: (i) modeling the behavior of a system into a formalism accepted by a model checking tool, (ii) specification of the desired properties in terms of temporal logic formulas and (iii) the model checking process. The verification process checks whether the model satisfies the properties represented in temporal logic formulas. In this thesis, we focus our work only on state-based model checking, which relies heavily on a Kripke model.

Let K be a Kripke model, Req be a set of formulas and Σ be a set of execution paths. The model checking process verifies whether every formula in Req is satisfied by the model. Formally, it checks whether $\forall f \in Req, \forall \pi \in \Sigma : K, \pi \vDash f$ is true or not. Further, we can also retrieve the set of execution paths, which satisfy the formulas in Req [23, 22].

As we can see, this approach can also be used to identify the fragment of the execution path as described above. One can also interpret that the formula is the *characterization* of the execution path, which should be identified by the refinement process.

However, the model checking process cannot be performed if the set of atomic propositions used to model the behavior of the system (the first step of the model checking) *differs* from the set of atomic propositions used to specify the desired properties (the second step of model checking). Therefore, we propose the combination of the temporal logic formalism with the description logic formalism to solve this problem.

Outline. We begin this chapter with Section 4.2 introducing the background. This involves the introduction of the description logic theory and temporal logic. We also present some related work with regard to our approach in description logic-based model checking in Section 4.3. Section 4.4 explains the concept of the description logic-based model checking. Section 4.5 defines a partially ordered relationship between temporal logic formula represented in description logic formalism, which is very useful to compare between two temporal logic formulas. Section 4.6 presents the algorithm to find the execution paths that are characterized by the temporal formula. Section 4.7 presents the proof that the algorithm always generates a finite set of traces. Section 4.8 presents the example of the description logic-based model checking. Section 4.9 concludes this chapter with a summary.

4.2 Background

The description logic-based model checking combines two formalisms, namely description logic and temporal logic. We briefly explain both formalisms as follows.

4.2.1 Description Logic

The description logic theory provides a formalism to represent domain-specific knowledge. It stems from the research of semantic networks [77], which was enriched with the logic foundation to achieve the desired formalism. This formalism defines a family of description logic language to represent the knowledge. Since the description logic-based knowledge representation inherits the semantic networks method, it has an advantage, which lies in the intuitive, easy and practical method of representing knowledge compared to the rule-based approach [7].

In this thesis we will only explain the description logic language, knowledge representation system and the reasoning services provided by the knowledge representation system. A more comprehensive discussion of description logic theory can be found in [7].

Description Logic Language

The description logic formalism defines a family of description logic language to represent the knowledge. The language is constructed by the language constructors. The most basic description logic language is the language class $\mathcal{AL}$, which can be extended by adding new constructors. This extension is

performed in order to achieve more expressiveness in representing the knowledge.

Representing the Knowledge

Since the DL-based knowledge representation originates from semantic network, both methods to represent knowledge are very similar. A semantic network can be roughly identified as a connected graph. The nodes and the edges of the graph represent the terminologies and the relationships between terminologies, respectively. In terms of DL-knowledge representation, the terminology and its relationship are referred to as *concept* and *role*, respectively. To avoid confusion with the *role* terminology, we shall use the term *relationship* instead.

In practice, DL language is used to describe the concepts, ranging from atomic concepts to complex concepts to build the knowledge. For example, the concepts represented by the DL language $\mathcal{AL}$ are constructed according to the following syntax rules [7]:

$$C, D \longrightarrow A|\top|\bot|\neg A|C \sqcap D|\forall R.D|\exists R.\top$$

C and D are arbitrary concepts and A is an atomic concept. In DL-knowledge representation, there always exist two default concepts, they are $\top$ and $\bot$, which serve as the most general and the most specific concepts that bound the taxonomy from the top and the bottom, respectively. The $\neg$ operator defines the complement of the concept. The $\sqcap$ binary operator denotes the intersection between two concepts. $\forall R.C$ constructor denotes that the concept has *only* relationships called R with the concept C. $\exists R.\top$ constructor means that the concept has *some* relationships called R with any arbitrary concept.

If we want to describe the term "rich employee" as employees, who only have expensive cars and have (at least) a house, then we can describe this as:

$$\mathsf{RichEmployee} \equiv \mathsf{Person} \sqcap \mathsf{Employee} \sqcap \forall \mathit{hasCar}.\mathsf{ExpensiveCar} \sqcap \exists \mathit{hasHouse}.\top.$$

DL-based Knowledge Representation System

A DL-based knowledge representation system consists of three parts.

- *terminology box* (Tbox)
  Intuitively, this container stores all definition of the terminologies or

concepts, such as the concept RichEmployee introduced above. It contains the domain specific knowledge, which is not bound to any particular real-world situation.

- *assertion box* (Abox)
  The purpose of this container is to store the knowledge about a particular real-world situation with respect to a Tbox. It contains *individuals* representing the real-world entities, which are asserted as a member of the concepts defined in the Tbox. For example, john is a rich person. Therefore, we insert the assertion RichPerson($john$) in the Abox.

- *reasoning service*
  The most important part of the DL-based knowledge representation system is its reasoning services. Currently, the actual list of DL-reasoner maintained by Sattler can be found on her website [84]. The most commonly used reasoning services are *subsumption* and *instance checking*. The first service checks whether a concept is more general than another concept. Formally, let $\mathcal{KB}$ be the knowledge base and C and D be the two concepts, which will be compared. According to knowledge base $\mathcal{KB}$, concept C is more general than concept D, if and only if, $\mathcal{KB} \vDash C \sqsupseteq D$. In other words, the knowledge base entails the statement $C \sqsupseteq D$. This reasoning service uses only the knowledge stored in Tbox to perform the reasoning process.

  The knowledge base $\mathcal{KB}$ has also an Abox containing the individuals, which are asserted to be the instances of particular concepts defined in Tbox. Therefore, the second reasoning service, instance checking, is very useful. For example, let x be an individual defined in the Abox of knowledge base $\mathcal{KB}$. In order to determine, whether x is an instance of concept C, we would perform instance checking. Formally, this process is written as $\mathcal{KB} \vDash C(x)$.This service uses both the Tbox and the Abox. In our work, we make use of the instance checking service to realize the model checking by using description logic reasoning engine.

Description Logic $\mathcal{SHK}$

We use the description logic language $\mathcal{SHK}$ to realize the description logic based model checking. This language is constructed from the base language $\mathcal{ALC}$ by adding some extensions. The first extension allows the definition of role transitivity. We call this language as $\mathcal{ALC}_{R+}$, which is often abbreviated as $\mathcal{S}$. The second extension allows the definition of hierarchy in relationships. Due to this extension the language is called $\mathcal{SH}$. Thus, $\mathcal{H}$ indicates the

$$
\begin{aligned}
\top^{\mathcal{I}} &= \Delta \\
\bot^{\mathcal{I}} &= \emptyset \\
A^{\mathcal{I}}, \mathcal{W} &= A^{\mathcal{I}} \\
R^{\mathcal{I}}, \mathcal{W} &= R^{\mathcal{I}} \\
(\neg A)^{\mathcal{I},\mathcal{W}} &= \Delta \setminus A^{\mathcal{I},\mathcal{W}} \\
(C \sqcap D)^{\mathcal{I},\mathcal{W}} &= C^{\mathcal{I},\mathcal{W}} \cap D^{\mathcal{I},\mathcal{W}} \\
(C \sqcup D)^{\mathcal{I},\mathcal{W}} &= C^{\mathcal{I},\mathcal{W}} \cup D^{\mathcal{I},\mathcal{W}} \\
(\forall R.C)^{\mathcal{I},\mathcal{W}} &= \{a \in \Delta \mid \forall b : (a,b) \in R^{\mathcal{I},\mathcal{W}} \rightarrow b \in C^{\mathcal{I},\mathcal{W}}\} \\
(\exists R.C)^{\mathcal{I},\mathcal{W}} &= \{a \in \Delta \mid \exists b : (a,b) \in R^{\mathcal{I},\mathcal{W}} \wedge b \in C^{\mathcal{I},\mathcal{W}}\} \\
(\mathbf{K}C)^{\mathcal{I},\mathcal{W}} &= \bigcap_{\mathcal{J} \in \mathcal{W}} C^{\mathcal{J},\mathcal{W}} = \{a \in \Delta \mid \forall \mathcal{J} \in \mathcal{W} : a \in C^{\mathcal{I},\mathcal{W}}\} \\
(\mathbf{K}R)^{\mathcal{I},\mathcal{W}} &= \bigcap_{\mathcal{J} \in \mathcal{W}} R^{\mathcal{J},\mathcal{W}} = \{a \in \Delta \mid \forall \mathcal{J} \in \mathcal{W} : (a,b) \in R^{\mathcal{I},\mathcal{W}}\}
\end{aligned}
$$

Figure 4.1: Formal semantics of $\mathcal{SHK}$

hierarchical relationships. The last extension allows the usage of epistemic operator **K** [31]. Thus, the language used in this thesis is called $\mathcal{SHK}$. We require the epistemic operator due to the fact that model checking relies on closed-world reasoning.

The formal semantics of description logic is determined by interpretation function $\mathcal{I}$ and interpretation domain Δ. The epistemic operator **K** defines an *epistemic interpretation* as a pair of $(\mathcal{I}, \mathcal{W})$. Under the common domain assumption and rigid term assumption, the set $\mathcal{W}$ contains the interpretation functions that are restricted only to the known assertions in the knowledge base. Let C and R be a concept and a relationship defined in description logic language. The interpretation function $\mathcal{I}$ maps each concept into a set of interpretation elements in Δ and each relationship into a set of binary relations $\Delta \times \Delta$. For example, the interpretation of concept C is $C^{\mathcal{I}} \subset \Delta$ and the interpretation of R is $R^{\mathcal{I}} \subset \Delta \times \Delta$. The complete formal semantics of $\mathcal{SHK}$ is shown in Figure 4.1.

Now let us discuss further the entailment semantics ($\models$) of description logic, which has already been introduced. We say that knowledge base $\mathcal{KB}$ entails the query $C \sqsupseteq D$ ($\mathcal{KB} \models C \sqsupseteq D$), if and only if, there exists a set of interpretation functions, so that for each interpretation function $\mathcal{I}$, each interpretation element of $D^{\mathcal{I}}$ is also an interpretation element of $C^{\mathcal{I}}$.

Formally, $\forall \mathcal{I} : C^{\mathcal{I}} \supseteq D^{\mathcal{I}}$.

4.2.2 CTL* logic

The computation tree logic-* (CTL*) is a class of temporal logics that defines the CTL* formula representing the desired property of the modeled system. Further, temporal logic is a derivation of the modal logic [52]. The CTL* formula is capable of expressing both linear-time and branching-time property of the execution path of the system. The formula is evaluated either over a path or on a state. Thus, CTL* formulas are categorized into *path formulas* and *state formulas*. Both path and state formulas have the standard boolean operators. Let f_i, g_i be the path formula and state formula, respectively. The CTL* formulas are constructed by using the following rules: A

$$\begin{aligned} g_1 \quad ::= \quad & f_1 | \neg g_1 | (g_1 \wedge g_2) | (g_1 \vee g_2) | \\ & (g_1 \mathbf{U} g_2) | \mathbf{X}\ g_1 | \mathbf{G}\ g_1 | \mathbf{F}\ g_1 \qquad (4.1) \\ f_1 \quad ::= \quad & p | \top | \neg f_1 | (f_1 \wedge f_2) | (f_1 \vee f_2) | \\ & \mathbf{A}\ [g_1] | \mathbf{E}\ [g_1]. \qquad (4.2) \end{aligned}$$

Figure 4.2: Syntax rules of the CTL* formula

path formula characterizes the linear-time property of an execution path by using the following temporal logic operators: **U** (until), **X** (next), **G** (holds globally) and **F** (finally). Complementary to this, a state formula characterizes the branching-time property of the execution paths that start from a state by using path quantifiers **A** (all possible paths) and **E** (some paths). Therefore, a state formula is constructed by adding the path quantifiers before the path formula. The CTL* formulas have more expressive power than both computation tree logic (CTL) and linear temporal logic (LTL) formulas combined. Thus, $\Sigma_{CTL*} \supset \Sigma_{CTL} \cup \Sigma_{LTL}$. A more comprehensive discussion of CTL* logic can be found in [24].

Let s and $\pi = s_0 s_1 s_2 \ldots$ be the state and the execution path in Kripke model K, respectively. The i-th state within the path π is denoted as π_i. Furthermore, let p be the atomic proposition, f_1, f_2 be the state formula and g_1 and g_2 be the path formula, respectively. The CTL* semantics are defined in Figure 4.3.

The equations in Figure 4.3 are inductively defined. Thus, some equations are defined by using the previously defined equations. Let us begin with the basic equation. Equation 4.3 says that state s satisfies the formula, which

$$
\begin{aligned}
s \vDash p &\Leftrightarrow p \in L(s) && (4.3)\\
s \vDash \neg f_1 &\Leftrightarrow s \nvDash f_1 && (4.4)\\
s \vDash f_1 \vee f_2 &\Leftrightarrow s \vDash f_1 \vee s \vDash f_2 && (4.5)\\
s \vDash f_1 \wedge f_2 &\Leftrightarrow s \vDash f_1 \wedge s \vDash f_2 && (4.6)\\
s \vDash \mathbf{E}\ (g_1) &\Leftrightarrow \text{ there exists a path } \pi \text{ starting with } s && \\
&\quad \text{such that } s \vDash g_1 && (4.7)\\
\pi \vDash f_1 &\Leftrightarrow \pi_0 \vDash f_1 && (4.8)\\
\pi \vDash \neg g_1 &\Leftrightarrow \pi \nvDash g_1 && (4.9)\\
\pi \vDash g_1 \vee g_2 &\Leftrightarrow \pi \vDash g_1 \vee s \vDash g_2 && (4.10)\\
\pi \vDash g_1 \wedge g_2 &\Leftrightarrow \pi \vDash g_1 \wedge s \vDash g_2 && (4.11)\\
\pi \vDash \mathbf{X}\ g_1 &\Leftrightarrow \pi_1 \vDash g_1 && (4.12)\\
\pi \vDash \mathbf{G}\ g_1 &\Leftrightarrow \forall i \in \mathbb{N}_0 : \pi_i \vDash g_1 && (4.13)\\
\pi \vDash \mathbf{F}\ g_1 &\Leftrightarrow \exists i \in \mathbb{N}_0 : \pi_i \vDash g_1 && (4.14)\\
\pi \vDash g_1 \mathbf{U} g_2 &\Leftrightarrow \exists j \in \mathbb{N}_0 : \pi_j \vDash g_2 \wedge && \\
&\quad \forall i \in \{0, \ldots, j-1\} : \pi_i \vDash g_1 && (4.15)
\end{aligned}
$$

Figure 4.3: CTL* semantics

consists only of proposition p, if and only if, the state s has the label named p.

The Equation 4.4, 4.9, 4.5, 4.10, 4.6 and 4.11 define the semantics of the common boolean logic operators in CTL* ($\neg, \vee, \wedge$).

Equation 4.4 and 4.9 say that state s or path π satisfies the *negation* of formula f or g, if and only if, s or π does not satisfy the formula f or g, respectively. Both equations determine the closed-world semantics of CTL*. Equation 4.5 (4.10) says that state s (path π) satisfies the disjunction ($\vee$) of formula f_1 and f_2 (g_1 and g_2), if and only if, state s (path π) satisfies one of the formulas, respectively. On the other hand, Equation 4.6 (4.11) says that state s (path π) satisfies the conjunction ($\wedge$) of formula f_1 and f_2 (g_1 and g_2), if and only if, state s (path π) satisfies both formulas, respectively.

Equation 4.7 and 4.8 determine the semantic entailments of a path formula from a state and vice versa. Equation 4.7 says that state s can satisfy *path* formula g_1, if and only if, there exists a path π starting with state s, so that $s \vDash g_1$. Further, Equation 4.8 says that path π can satisfy *state* formula f_1, if and only if, the first state of path π satisfies state formula f_1.

The next four equations (Equation 4.12, 4.13, 4.14 and 4.15 define the

semantics of temporal logic operators. Equation 4.12 says that path π satisfies formula $\mathbf{X}\ g_1$, if and only if, the path starting from state pi_1 satisfies g_1. This operator is also called the "Next" operator, since the next state after the first state satisfies g_1.

Equation 4.13 says that path π satisfies formula $\mathbf{G}\ \ g_1$, if and only if, all paths starting from each state within path π satisfy formula g_1. Clearly, π must be a path with infinite length in order to satisfy this condition. This operator is also called the "Globally" operator, since the fulfillment of g_1 must hold in every sub path of π.

Equation 4.14 says that path π satisfies formula $\mathbf{F}\ \ g_1$, if and only if, there exists a sub path in π, which satisfy formula g_1. It should be noted that $\mathbf{F}$ operator is the complement of $\mathbf{G}$ operator. This operator is also called "Future" operator, since the fulfillment of formula g_1 must only hold in a future sub path of π.

Equation 4.15 says that path π satisfies $g_1 \mathbf{U} g_2$, if and only if, all sub paths of π from the first state to a certain state satisfy g_1 and a sub path starting immediately after this state satisfies g_2. This operator is also called "Until" operator, since formula g_1 must always hold, until formula g_2 holds.

Lastly, it should be noted that $\mathbf{A}\ f = \neg\mathbf{E}\ \neg f$, since $\mathbf{A}$ is the complement of $\mathbf{E}$.

Model Checking Model checking is a very useful technique, which is often used in the system verification process. It is an automated process that verifies whether a system satisfies certain requirement specification. Typically, the behavior of the system is modeled as a Kripke model or finite state machine and the requirements specification is written as a set of formulas. Formally, the behavior of the system is represented by model K and the requirements specification is written as a set of formulas Req. Thus, model checking finds the set $\{s \in W \mid \forall f \in Req :\ K, s \vDash f\}$, where for each $s \in I$, s fulfills all formulas $f \in Req$. This performs model checking of the path formula. On the other hand, if we replace s, I and f with π, Σ and g, respectively, then we perform model checking of the state formula.

The fulfillment relationship ($\vDash$) is derived from the formal semantics presented in Figure 4.3.

4.3 Related Work

In this section we present several approaches, which either are useful to our approach or have a similar motivation to ours. The first approach, which is presented in Section 4.3.1, extends the description logics semantics with

temporal semantics. The big disadvantage of this approach is its lacks of proof of concept. There exists no implementation of the reasoning engine for this class of description logic language. Therefore, we cannot use this approach to support our work.

The second and the third approaches are presented in Section 4.3.2. These approaches address the matchmaking process of the web service behavior. The matchmaking process aims at finding a pair of web services, which have similar behavior. The second approach uses the intersection of two automata to determine whether two web services have similar behavior or not. The third approach uses the graph matching algorithm to realize the matchmaking process. Although the implementations of these two last approaches exist, they have a disadvantage. It is very difficult to match the service against a specification. In these approaches, the specification should be written either as an automaton or a graph, which is very impracticable. Furthermore, these two approaches do not support a matchmaking process, which is based on ontology. Therefore, a matchmaking process between two web services from different domains is not possible. This implies that we cannot use these approaches to perform the pattern matching, which heavily relies on the ontology.

4.3.1 Introducing Temporal Operator to the Description Logics

Basically, this approach tries to combine temporal logic formalism with description logic formalism. However, the idea to integrate temporal logic formalism with description logics formalism is not new. Several works [88, 89] attempt to integrate temporal logic formalism by extending the description logic formalism. In contrast, our approach combines both formalisms by expressing the temporal logic formalism in terms of the description logics language, which has the description logic language formalism. As a simple case, we consider only the restricted CTL* temporal logic. It is clear that the previous works aim for the temporal extension of description logic formalisms, which have more expressive power than our work.

The work of Baader et al. [8] incorporates action formalism to represent the dynamic behavior of a system. Again, we use another approach by using the Kripke model to represent the dynamic behavior of a system.

Another approach which is very relevant to us is the work of Ben-David et al. [13]. In their work, the state transitions model, which is a transformation of the Kripke model, is represented as a set of the Tbox concept definitions. In our work, the Kripke model is represented as a set of assertions in the

Abox. Furthermore, we consider the ontology of the atomic propositions in our work.

4.3.2 Matchmaking of the Behavior of Web Services

Matchmaking of the behavior of web services [45, 58] is a very useful mechanism in finding the desired web services, whose behaviors comply with the requirements of service consumers. If we represent the behavior of the web service and the service consumer's requirements as a finite state machine and temporal logic formulas, respectively, we claim that the matchmaking problem is similar to the model checking problem. Thus, we can say that the matchmaking problem is also similar to the pattern matching problem. In this sub section we present several approaches that deal with the matchmaking problem. These approaches address the problem using different methods.

Grigori et al. presented in [45] an approach to compare and match web services behaviors by using a graph matching algorithm. In their approach, they specify the requirements of the service consumers and the behavior of the web service as two separate graphs. By using the existing graph matching algorithm, they aim at solving the matchmaking problem.

Mahleko presented in his Ph. D. thesis [58] the approach that addresses the same problem, namely matchmaking of the webservices behaviors. However, he used a different approach to solve this problem. In his approach, he represents both the web service behavior and the consumer's requirements as finite state automata. If the intersection of these two finite state automata is not empty, then the web service behavior matches the consumer's requirements.

4.4 Description Logic-based Model Checking

Recall the model checking process, which is discussed in Section 4.2.2. We have already stated in our motivation that we want to use the reasoning service of the description logic-based knowledge base in order to perform the model checking. Therefore, we propose an approach to establish the description logic-based model checking. The main idea behind this approach is based on the facts stating that: (i) a class of temporal logic formula (LTL formula) can be translated into a finite automaton [24, p. 121] and (ii) a finite automaton can be represented as a set of axioms [7, p. 90].

We propose an approach to solve the problem motivated in Section 4.1 by constructing the ontology of the atomic propositions, which are used in the model checking process. Thus, the ontology can be used to reason whether

an atomic proposition occurring in the formula is equal or more general to another atomic proposition occurring in the Kripke model.

To integrate the ontology reasoning process in the model checking process, we represent the Kripke model in the description logic-based knowledge base and represent a subset of CTL* semantics in terms of description logic language, which has the description logic semantics. Our intention is to perform the model checking process by using the reasoning service of the DL-based knowledge base. Obviously, the DL-based knowledge base also supports the ontology reasoning process. Thus, we can perform the model checking process together with the ontology reasoning process only by using the reasoning service of the DL-based knowledge base.

We intend to perform the model checking process by using *instance checking* reasoning service, which has already been discussed in Section 4.2.1. In our case, we want the model checking

$$K, s \models f$$

to be equivalent to

$$\mathcal{KB} \models D(x),$$

which is the instance checking query.

It should be noted that the entailment ($\models$) of the model checking has a different semantics compared to the instance checking. The semantics of the model checking entailment can be found in Figure 4.3, whereas the semantics of description logic reasoning, particularly instance checking, can be found in Figure 4.1. Therefore, if the entailment symbol ($\models$) appears immediately after a Kripke model and a state (for example, K, s), then it has the CTL* semantics. On the other hand, if the entailment symbol appears immediately after a knowledge base (for example, $\mathcal{KB}$), then it has the description logic semantics.

The knowledge base $\mathcal{KB} : \langle \mathcal{T}_{\mathcal{KB}}, \mathcal{A}_{\mathcal{KB}} \rangle$ consists of $\mathcal{T}_{\mathcal{KB}} = \mathcal{T}_{AP} \cup \mathcal{T}_{ONT} \cup \mathcal{T}_f$ and $\mathcal{A}_{\mathcal{KB}}$. The constructions of the Tboxes and Abox are presented in the next subsections. $\mathcal{T}_{AP}$ contains the concepts, which represent the atomic propositions in AP. $\mathcal{T}_{ONT}$ contains the ontology that represent the equivalence and generalization relationships between the atomic propositions. $\mathcal{T}_f$ contains the concepts representing the temporal formulas. $\mathcal{A}_{\mathcal{KB}}$ is the assertion box representing the Kripke model. The concept D represents the formula f and the individual x represents the state s.

The currently known description logic reasoner that supports epistemic operator is only the Pellet reasoner [74]. However, the reasoner still has some restrictions:

- It only allows the appearance of the epistemic operator in the query language.
  For example, the instance checking $\mathcal{KB} \models D(x)$ is actually a query operation. Thus, the Pellet reasoning engine would only allow the occurrence of epistemic operator $\mathbf{K}$ in concept D and not in any concepts stored in knowledge base $\mathcal{KB}$.

- It does not support relationship restriction having epistemic operator $(\forall \mathbf{K}R.D)$.
  According to the theoretical definition of the epistemic operator $\mathbf{K}$, the operator can appear immediately before a concept $(\mathbf{K}C)$, or inside an existential quantifier $(\exists \mathbf{K}R.D)$ or inside a relationship restriction $(\forall \mathbf{K}R.D)$. Due to the limitation of the implementation, Pellet does not allow the occurrence of epistemic operator inside a relationship restriction.

Due to these limitations, we will focus only on a subset of CTL* formulas that has the form of $\mathbf{E}\,[\phi]$, where ϕ is a LTL formula. For example, we only allow such formula: $f = \mathbf{E}\,(\mathit{check\_loan\_appl} \wedge \mathbf{F}\ \mathit{approve\_loan\_appl})$.

Our approach to realize description logic-based model checking is divided into the following steps:

1. representing the Kripke model in Aboxes (constructing $\mathcal{A}_{\mathcal{KB}}$),
2. translating the formulas into concepts (constructing $\mathcal{T}_f$),
3. defining the CTL* semantics in terms of $\mathcal{SHK}$ description logic language semantics and
4. defining the ontology of atomic propositions (constructing $\mathcal{T}_{AP}$ and $\mathcal{T}_{ONT}$).

These steps are depicted in Figure 4.4. The left side of this figure shows the components required for the traditional model checking process. The right side of this figure shows the components required for the model checking process, which is performed by using the description logic-based knowledge base. The arrows denote the translations of the components. We propose these translations and the construction of the ontology of atomic propositions in order to enable description logic-based model checking. These steps are explained in the next sections.

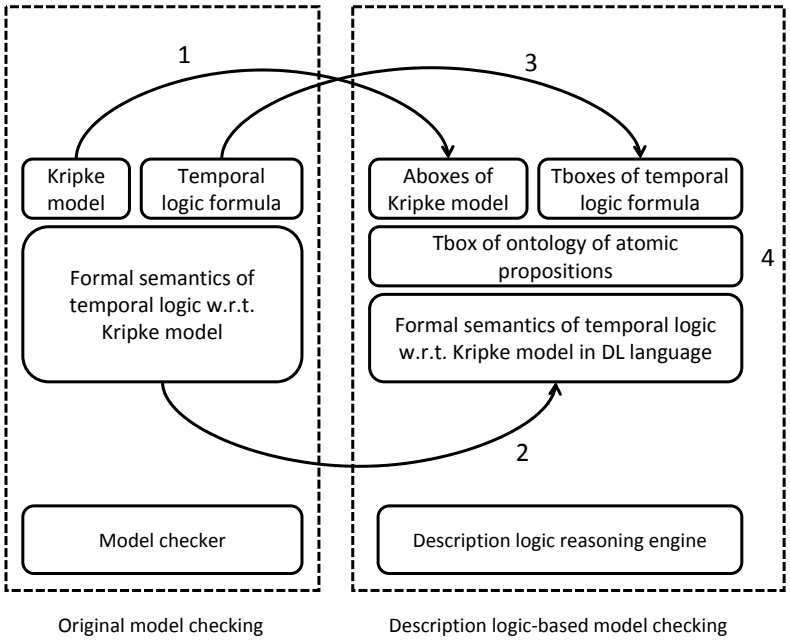

Figure 4.4: Translation of the components required in model checking

4.4.1 Representation of the Kripke Model as ABoxes

The Kripke model is represented as a set of assertions, which is contained in Abox called $\mathcal{A}_{\mathcal{KB}}$. $\mathcal{A}_{\mathcal{KB}}$ is divided into two Aboxes, they are $\mathcal{A}_{RL}$ and $\mathcal{A}_L$.

$\mathcal{A}_{RL}$ contains the assertions about state transitions that correspond to set RL of Kripke model K. For example, the assertion $\mathit{next}(x_i, x_j)$ represents the state transitions between states s_i and s_j of the Kripke model K. Note that the individuals x_i and x_j represent the states s_i and s_j of the Kripke model, respectively.

$\mathcal{A}_L$ contains the assertions about the labels of each state. For example, the assertion $V(x_j)$ represents the presence of label p in the state s. Thus, this assertion is defined for each $p \in L(s)$. Note that the concept V represents the collection of individuals representing the states that have the label p.

4.4.2 Formal Semantics of Restricted CTL* Formulas in Description Logics

In this section, we propose the representation of CTL* logic semantics in terms of DL language family $\mathcal{SHK}$ semantics. As previously stated at the beginning of Section 4.4, we aim at using the instance checking reasoning to

perform the model checking process. Since we want to perform finite model checking (closed-world assumption) that enables negation-as-failure property [33], we require the usage of epistemic operator **K** that assumes the common domain and rigid term of the knowledge base.

We only consider the restricted CTL* formulas, which is defined as follows:

$$\begin{aligned} g_1 &::= p|\top|\neg p|(g_1 \wedge g_2)|(g_1 \vee g_2)| \\ &\quad (g_1 \mathbf{U} g_2)|\mathbf{X}\ g_1|\mathbf{G}\ g_1|\mathbf{F}\ g_1 \qquad (4.16) \\ f_1 &::= \mathbf{E}\ [g_1]. \qquad (4.17) \end{aligned}$$

Figure 4.5: Syntax rules of the restricted CTL* formula

It should be noted that the quantifier (**E**), temporal ($\mathbf{U}, \mathbf{F}, \mathbf{G}, \mathbf{X}$) and boolean ($\neg, \wedge, \vee$) operators are the same with the operators of the CTL* formulas. The difference lies in the path quantifier and the negation. We only allow the existential path quantifier (**E**) and the occurrence of negation immediately before an atomic proposition.

Recall the definitions of p, g_1, g_2, s, π and $\mathcal{KB}, x$ from Section 4.2.2 and the first part of Section 4.4, respectively. Let D_i and D_{aux} and be the concept representing the sub formula g_i and the additional auxiliary concept, respectively. The symbol σ denotes a sequence of Abox individuals, which is the description logics representation of trace π. Thus, σ_i denotes the Abox individual at the index i.

We propose the translation of the CTL* logic semantics to $\mathcal{SHK}$ semantics that are defined in Figure 4.6. The complete proofs of this translation can be found in Appendix A. The left hand of the equations are the semantics of CTL* and the right hand of the equations are the corresponding translations, which are represented in description logic language semantics.

The purpose of the epistemic operator, which is used to represent CTL* semantics in terms of description logic semantics, is to enable the closed-world reasoning. Closed-world reasoning facilitates the evaluation of failure as negation. To understand the meaning of the failure as negation, let us consider the formal semantics of CTL* showed in Figure 4.3. Based on CTL* semantics, we conclude that $s \nvDash p$ is equivalent to $s \vDash \neg p$. This means that the failure of the entailment is equivalent to the negation.

In description logic representation, we define

$$s \nvDash p \qquad (4.28)$$

$$s \vDash \neg p \qquad (4.29)$$

$$
\begin{aligned}
s \vDash p_i &\Leftrightarrow \mathcal{KB} \vDash \mathbf{K}V_i(x) && (4.18)\\
s \vDash \neg p_i &\Leftrightarrow \mathcal{KB} \vDash \neg\mathbf{K}V_i(x) && (4.19)\\
s \vDash \mathbf{E}\ (g_1) &\Leftrightarrow \mathcal{KB} \vDash D_1(x) && (4.20)\\
\pi \vDash g_1 &\Leftrightarrow \mathcal{KB} \vDash D_1(\sigma_0) && (4.21)\\
\pi \vDash g_1 \vee g_2 &\Leftrightarrow \mathcal{KB} \vDash (D_1 \sqcup D_2)(\sigma_0) && (4.22)\\
\pi \vDash g_1 \wedge g_2 &\Leftrightarrow \mathcal{KB} \vDash (D_1 \sqcap D_2)(\sigma_0) && (4.23)\\
\pi \vDash \mathbf{X}\ g_1 &\Leftrightarrow \mathcal{KB} \vDash (\exists\mathbf{K}\mathit{next}.D_1)(\sigma_0) && (4.24)\\
\pi \vDash \mathbf{G}\ g_1 &\Leftrightarrow \langle \mathcal{T}_{\mathcal{KB}} \cup \{D_{aux} \equiv D_1 \sqcap \exists\mathbf{K}\mathit{next}.D_{aux}\}, \mathcal{A}_{\mathcal{KB}}\rangle && \\
&\quad \vDash D_{aux}\ (\sigma_0) && (4.25)\\
\pi \vDash \mathbf{F}\ g_1 &\Leftrightarrow \mathcal{KB} \vDash (D_1 \sqcup \exists\mathit{future}.D_1)\ (\sigma_0) && (4.26)\\
\pi \vDash g_1\mathbf{U}g_2 &\Leftrightarrow \langle \mathcal{T}_{KB} \cup \{D_{aux} \equiv D_2 \sqcup (D_1 \sqcap \exists\mathbf{K}\mathit{next}.D_{aux} \sqcap \exists\mathit{future}.D_2)\} && \\
&\quad , \mathcal{A}_{\mathcal{KB}}\rangle \vDash D_{aux}\ (\sigma_0) && (4.27)
\end{aligned}
$$

Figure 4.6: Formal semantics of a subset of CTL* logic in $\mathcal{SHK}$

as

$$\mathcal{KB} \nvDash V(x) \tag{4.30}$$
$$\mathcal{KB} \vDash \neg V(x), \tag{4.31}$$

respectively. The individual x represents the state s and the concept V represents the set of individuals having the label p. Equation 4.30 states that it could not be deduced from the knowledge base that x is a member of V. Equation 4.31 states that the individual x is *not* a member of the concept V. In the CTL* semantics, both equations are equivalent.

However, in description logic semantics, Equation 4.30 and Equation 4.31 are not equivalent, although both equations represent Equation 4.28 and Equation 4.29, respectively. This is due to the fact that description logic semantics have the open-world reasoning property. This means that if the knowledge base cannot deduce that x is a member of V, it does not necessarily imply that x is *not* a member of V. Since the knowledge base does not have sufficient knowledge to answer the query, it cannot deduce the fact in answering the query.

Therefore, we introduce the epistemic operator $\mathbf{K}$ in order to have the closed-world reasoning property in the description logic reasoning. The epistemic operator $\mathbf{K}$ immediately appears in the atomic proposition and the

state transition. (see equations 4.18, 4.19, 4.24 - 4.25 and 4.27.)

Expressing the semantics of modal operators **X** and **F** in description logic is very simple, since we use *next* relationship and transitive relationship *future* $\in R_+$ to build the semantics. It should be noted that the relationship *next* is a sub-relationship of *future*. Thus, we define the transitivity of *future* and the assertion *next* $\sqsubseteq$ *future* in the Tbox.

To express the semantics of modal operator **G**, we use the cyclic concept definition for expressing the models of infinite traces that satisfy the **G** operator. Since the Tbox contains cyclic concept definitions, we require a particular interpretation semantics, which are used in the reasoning engine.

It should be noted that the description logic language formalism has three different types of interpretation semantics [68]. They are: *least-fixpoint* semantics, *greatest fixpoint* semantics and *descriptive* semantics. The least (greatest) fixpoint semantics consider the smallest (largest) set of models with cyclic relationships, which satisfy cyclic concept definitions. The descriptive semantics, however, do not consider the models with cyclic relationships [7]. Thus, the descriptive semantics can not be used to represent the semantics of **G** modal operator, which is only satisfied by a cyclic model containing some infinite traces.

If the Tbox does not contain any cyclic concept definition, then these interpretation semantics are all the same. Thus, the choice of the interpretation semantics used in the reasoning engine will not affect the reasoning result [96]. The descriptive semantics are usually used in most reasoning engines, due to their simplicity of implementation.

However, if the Tbox contains some cyclic concept definitions, then these three interpretation semantics with regard to the cyclic concepts differ from each other [96]. Baader points out that the least-fixpoint semantics always consider a set of empty models, which satisfy the cyclic concepts' interpretation [6]. Such interpretation semantics are uninteresting to us. On the other hand, the greatest-fixpoint semantics can interpret cyclic concept definitions as the largest set of models, containing the cyclic relationships. This suits our need to correctly interpret the **G** modal operator. Therefore, we require that the reasoning engine should support the greatest-fixpoint semantics in order to correctly represent the semantic of **G** modal operator.

Although the translation of the semantic of **U** operator uses cyclic concept definition, we can use descriptive semantics to correctly capture the meaning of **U** operator. This is due to the fact that **U** operator can be satisfied by finite traces.

Soundness and Correctness of the CTL* Logic. The most important properties of any logic are the *soundness* and *completeness* properties. In a nutshell, these properties assure that the truth values of the derivation of any formula, which is either derived using natural deduction ($\vdash$) or interpretation model ($\models$), are equivalent. In [60] soundness and correctness are defined as follows:

Definition 23 (Soundness) *A formal system FS is sound, if and only if for each set of formulas Z and every formula A:*

$$Z \vdash A \Rightarrow Z \models A.$$

This definition says that every A, which is derivable from Z using the deduction rules of the corresponding logic, is also satisfied by the interpretation models, which satisfy all formulas in Z.

Definition 24 (Completeness) *A formal system FS is complete, if and only if, for each set of formulas Z and every formula A:*

$$Z \models A \Rightarrow Z \vdash A.$$

Conversely to the Definition 23, this definition says that every A, which is satisfied by the interpretation models satisfying all formulas in Z, is derivable from Z by using the deduction rules of the corresponding logic.

The original interpretation model of restricted CTL* is namely Kripke model. In this thesis, we represent the Kripke model in terms of Description Logic language and use the Description Logic inference mechanism to perform the logical entailment ($\models$) of the formal semantics of restricted CTL*. In other words, we try to perform the entailment of restricted CTL*, which is based on Kripke model, by using the Description Logic representation and its inference mechanism.

As we can see in Figure 4.6, each semantic entailment of restricted CTL* is *semantically equivalent* to the corresponding concept represented in Description Logic according to the inference mechanism (instance checking). Therefore, we say that we *emulate* the formal semantics of restricted CTL* by using Description Logic representation and its inference mechanism. This implies that our Description Logic-based representation of the semantics of restricted CTL* also preserve the soundness and the correctness of restricted CTL*.

4.4.3 Translating CTL* Formulas into Tbox Concepts

Recall that the restricted CTL* formulas has the form of $\mathbf{E}\ [\phi]$, where ϕ is the LTL formula enclosed in a $\mathbf{E}$ path quantifier. Therefore, we only translate the LTL formula ϕ into Tbox concepts. Further, the negation occuring inside the LTL formula only appears beside the atomic propositions. The first step in this translation is to decompose ϕ into a parse tree.

Definition 25 *A parse tree T_f of the formula f is a binary tree, whose nodes represent the sub formula of f except the root node, which represents the formula f itself.*

Now we can perform the translation of the LTL formula ϕ, by visiting the nodes in the parse tree starting from the leaf nodes and toward the root node. Suppose that we have a counter idx, which has the initial value 0. For each visit in the node, we create a new axiom in the Tbox $\mathcal{T}_f$ according to the rules in Figure 4.7 and increase the counter idx. The concepts D_i and D_j refer to the concepts created after translating the previous sub formulas g_i and g_j, respectively.

LTL formula	concepts added to $\mathcal{T}_f$
p	$D_{idx} \equiv \mathbf{K}V$
$\neg p$	$D_{idx} \equiv \neg\mathbf{K}V$
$g_i \vee g_j$	$D_{idx} \equiv D_i \sqcup D_j$
$g_i \wedge g_j$	$D_{idx} \equiv D_i \sqcap D_j$
$\mathbf{X}g_i$	$D_{idx} \equiv \exists\mathbf{K}\mathit{next}.D_i$
$\mathbf{G}g_i$	$D_{idx} \equiv D_i \sqcap \exists\mathbf{K}\mathit{next}.(D_{idx} \sqcup D_i)$
$\mathbf{F}g_i$	$D_{idx} \equiv D_i \sqcup \exists\mathit{future}.D_i$
$g_i\mathbf{U}g_j$	$D_{idx} \equiv D_j \sqcup (D_i \sqcap \exists\mathbf{K}\mathit{next}.D_{idx} \sqcap \exists\mathit{future}.D_j)$

Figure 4.7: Translation table of a subset of CTL* sub formulas into $\mathcal{SHK}$ concepts

It is worth noting that the concept generated by translating $\mathbf{G}\ g_i$ according to Figure 4.7 differs from the concept defined in the Equation 4.25, which is semantically equivalent to $\mathbf{G}\ g_i$. The concept generated by the translation rule represents the formula, which accepts both finite and infinite traces of $\mathbf{G}\ g_i$. This is due to the fact that the instances of the instances of concept $Y \equiv D_i \sqcap \exists\mathbf{K}\mathit{next}.(Y \sqcup D_i)$ have a *next* connection either to an instance of Y or of D_i.

On the other hand, the concept defined in Equation 4.25 accepts only infinite traces. This is due to the fact that the instances of concept $X \equiv$

$D_i \sqcap \exists \mathbf{K}\mathit{next}.X$ have only a *next* connection to an instance of X, which also has a connection to an instance of X. Thus, only infinite traces are accepted.

4.4.4 Ontology of Atomic Propositions

The ontology of atomic propositions $\mathcal{T}_{ONT}$ provides the common vocabulary that aims at bridging the semantic gap, which occurs during the model checking process. The semantic gap exists when the Kripke model and the temporal logic formula use different vocabularies of atomic propositions.

In order to identify this problem, let us consider Equation 4.3 ($s \vDash p \Leftrightarrow p \in L(s)$) in Page 41, which is the semantic of CTL*. According to the equation, state s satisfies formula p ($s \vDash p$), if and only if, state s has a label called p, which appears in the formula ($p \in L(s)$). A problem arises if we use another atomic proposition to represent the formula. Consider Example 12 that explains this problem in more detail.

Example 12 *Let Kripke model $K : \langle W = I = \{z\}, RL = \emptyset, L = \{z \mapsto \{start\}\}, AP = \{start, Anfang\}\rangle$ and formula $f = Anfang$. According to CTL* semantics, we can see that $z \nvDash Anfang$, since $start \neq Anfang$ and $Anfang \notin L(z)$, although we interpret that the atomic proposition start is equivalent to $Anfang$. Therefore, the original model checking technique cannot fulfill our requirements.*

We solve this problem by introducing *semantic bridges* between atomic propositions *start* and $Anfang$. This is possible, since we perform the model checking by using the description logic reasoning engine. First, we define the following Tboxes, $\mathcal{T}_{AP}$ and $\mathcal{T}_{ONT}$.

Definition 26 ($\mathcal{T}_{AP}$) *Tbox $\mathcal{T}_{AP}$ is constructed by defining a concept for each atomic proposition in AP. The names of these concepts are uniquely defined.*

The above definition states that Tbox $\mathcal{T}_{AP}$ contains only the concepts, which represent the atomic propositions. Since the name of the concepts should be unique, the mapping between AP and $\mathcal{T}_{AP}$ is an injection.

Definition 27 ($\mathcal{T}_{ONT}$) *Tbox $\mathcal{T}_{ONT}$ contains the equivalence ($\equiv$) and the generalization ($\sqsupset$) between two concepts in $\mathcal{T}_{AP}$, which represent two different atomic propositions.*

Let C_1 and C_2 be the concepts in $\mathcal{T}_{AP}$. If the two atomic propositions represented by C_1 and C_2 are equivalent, then $C_1 \equiv C_2$ is defined in $\mathcal{T}_{ONT}$. On the other hand, if the atomic proposition represented by C_1 is more general

than the atomic proposition represented by C_2, then $C_1 \sqsupseteq C_2$ is defined in $\mathcal{T}_{ONT}$.

We then construct the following proposition:

Proposition 1 *Let $p \in L(s) \subset AP$, $q \in AP$ and $V, Q \in \mathcal{T}_{AP}$. x, V, Q and $\mathcal{T}_{AP}$ represent s, p, q and AP,respectively. Thus, $s \vDash q$ if and only if, $\mathcal{KB} \vDash Q \sqsupseteq V \wedge \mathcal{KB} \vDash \mathbf{K}V(x)$.*

Proof 1 **"⇒":** *We will use an indirect proof. Let us assume that $\mathcal{KB} \nvDash Q \sqsupseteq V \vee \mathcal{KB} \nvDash \mathbf{K}V(x)$. $\mathcal{KB} \nvDash \mathbf{K}V(x)$ is impossible, since according to Equation 4.18 $p \in L(s)$ is equivalent to $\mathcal{KB} \vDash \mathbf{K}V(x)$. Thus, only $\mathcal{KB} \nvDash Q \sqsupseteq V$ may be true. However, this implies that $\mathcal{KB} \nvDash \mathbf{K}Q(x)$, because (i) the Abox of $\mathcal{KB}$ does not define the assertion "$Q(x)$", but only defines the assertion "$V(x)$" and (ii) $\mathcal{KB} \nvDash Q \sqsupseteq V$. Thus, according to Equation 4.18, $\mathcal{KB} \nvDash \mathbf{K}Q(x)$ is equivalent to $s \nvDash q$, which contradicts the assumption.*

"⇐": *According to description logics semantics, the statement $\mathcal{KB} \vDash Q \sqsupseteq V \wedge \mathcal{KB} \vDash \mathbf{K}V(x)$ implies that $\mathcal{KB} \vDash \mathbf{K}Q(x)$. Furthermore, according to the Equation 4.18, it is equivalent to $s \vDash q$.*

This proposition states that although p, which is the atomic proposition occurring in the Kripke model, and q, which is the atomic proposition occurring in formula, are different atomic propositions, which have the same interpretation, the model checking can be performed. This is possible, since $\mathcal{T}_{ONT}$ provides the *semantic* bridge between these atomic propositions. This kind of *semantic* bridge is only facilitated in the description logic-based model checking.

Now let us consider the following example, which explains the purpose of the ontology of atomic propositions in more detail:

Example 13 *Let Kripke model $K : \langle W = I = \{z\}, RL = \emptyset, L = \{z \mapsto \{start\}\}, AP = \{start, Anfang\}\rangle$ and formula $f = Anfang$. According to the translations defined Section 4.4.1 and 4.4.3, we have knowledge base $\mathcal{KB}$ representing Kripke model K and formula f, which is defined as follows:*
$\mathcal{KB} : \langle \mathcal{A}_{\mathcal{KB}} = \{\mathsf{C\_Start}(x)\}, \mathcal{T}_{\mathcal{KB}} = \{\mathsf{C\_Start}, \mathsf{C\_Anfang}, \mathsf{C\_Start} \equiv \mathsf{C\_Anfang}\}\rangle$.

According to $\mathcal{T}_{\mathcal{KB}}$, we have $\mathcal{KB} \vDash \mathsf{C\_Anfang} \sqsupseteq \mathsf{C\_Start}$. Further, due to assertion $\mathsf{C\_Start}(x)$, we have $\mathcal{KB} \vDash \mathbf{K}\mathsf{C\_Start}(x)$. Thus, according to Proposition 1, this is equivalent to $s \vDash Anfang$.

From Example 13, we can see that by using the description logic-based model checking and the appropriate ontology, in this case $\mathsf{C\_Start} \equiv \mathsf{C\_Anfang}$, we can derive that s satisfies formula $Anfang$. Obviously, the ontology expresses our interpretation of atomic propositions, which says that *start* is equivalent to $Anfang$.

4.5 Relationship between Temporal Logic Formulas

The representation of the temporal logic formula, especially the subset of CTL* formulas, as a description logic concept allows us to use the subsumption relationship and thus to define a partial order relation $\prec$ between the temporal logic formulas, especially a subset of CTL* formulas.

Definition 28 *Let g_1 and g_2 be the restricted CTL* formulas. We define the relation $g_1 \prec g_2$ if and only if, $\forall \pi \in \Psi : (\pi \vDash g_1 \Rightarrow \pi \vDash g_2)$. Ψ is the non-empty set of the possible execution paths.*

Proposition 2 *Let D_{g_1} and D_{g_2} be the $\mathcal{SHK}$ concepts representing the formulas g_1 and g_2, respectively. $\mathcal{KB} \vDash D_{g_1} \sqsubset D_{g_2}$ if and only if, $g_1 \prec g_2$.*

Proof 2 **"⇒":** *According to the description logics semantics, $\mathcal{KB} \vDash D_{g_1} \sqsubset D_{g_2}$ implies that $\forall x \in IN : (\mathcal{KB} \vDash D_{g_1}(x) \Rightarrow \mathcal{KB} \vDash D_{g_2}(x))$. With the formal semantics defined in Figure 4.6 we can construct that $\forall \pi \in \Psi : (\pi \vDash g_1 \Rightarrow \pi \vDash g_2)$. Therefore, according to Definition 28, $g_1 \prec g_2$.*
"⇐": *According to Definition 28, $g_1 \prec g_2$ implies that $\forall \pi \in \Psi : (\pi \vDash g_1 \Rightarrow \pi \vDash g_2)$. By using the formal semantics defined in Figure 4.6, we can prove that $\forall x \in IN : (\mathcal{KB} \vDash D_{g_1}(x) \Rightarrow \mathcal{KB} \vDash D_{g_2}(x))$.*

Such a relationship is very useful for comparing different policies specified as temporal logic formulas. For example, one may want to find out whether a policy is more general than the other policies. This can be done by performing the subsumption reasoning w.r.t. the Tbox.

4.6 Finding Traces of a formula

A trace[1] of the restricted CTL* temporal formula is the sequence of states that satisfies the corresponding formula. In this section we present an approach, which finds the witness of the restricted CTL* formula by using the instance retrieval reasoning. The instance retrieval reasoning is another form of instance checking. It collects a set of individuals, which are members of a particular concept. It can be performed by executing instance checking for each individual stored in the knowledge base.

We are interested only in finding traces that have a *finite* trace [36, 59]. Furthermore, the set of traces generated by this process is always finite, as we will see later.

[1] also known as a path or a witness

The construction of the traces employs the post-processing recursion technique. Figure 4.8 shows the algorithm. The algorithm takes variable n, which is from the datatype `node`. This datatype represents the node of the parse tree. Thus, each node has at most two references to other nodes. To retrieve the traces of a formula, this algorithm takes the root of the parse tree T_f as its input. The algorithm ends by returning a set of traces S.

4.6.1 Additional Functions

In this section we will explain the sub functions used in the algorithm shown in Figure 4.8. Some of these sub functions find the set of traces by employing the graph traversal algorithm. Each visited state is marked to avoid *redundant cycles* in the traces. The functions are explained as follows:

- `getStates(`K`)`
  This function takes Kripke model K as its parameter and returns a set of states of the model K.

- `getSubFormula(`n`)`
  This function takes node n of the parse tree as its parameter and returns the sub formula represented by this node.

- `nodeType(`n`)`
  This function takes node n of the parse tree as its parameter and returns the type of the nodes, which are atomic proposition, $\wedge$, $\vee$, **X**, **G**, **F** and **U**.

- `concat(`π_1, π_2`)`
  This function takes two traces π_1 and π_2 as its parameters and returns the concatenation of π_1 followed by π_2.

- `head(`π`)`
  This function takes trace π as its parameter and returns the first state of the trace.

- `findTraceX(`s_i, s_j, K, f`)`
  This function takes two states s_i and s_j, model K and sub formula f as its parameters. If s_i is succeeded by s_j, this function returns a trace consisting only of s_i. Otherwise, it returns an empty set.

- `findTraceG(`s_i, s_j, K, f`)`
  This function takes two states s_i and s_j, model K and sub formula f as its parameters. It returns a set of all traces within the model

```
   Input: Node n of the parse tree
   Input: Kripke Model K
 1 tr_1 ← ∅ ; tr_2 ← ∅ ; tr_out ← ∅ ;
 2 switch nodeType(n) do
 3     case atomicProposition
 4         foreach s ∈ getStates(K) ∧ K, s ⊨ getSubFormula(n) do
 5             tr_1 ← tr_1 ∪ {s} ;                          /* add a one-state trace */
 6     case ∧-node
 7         tr_1 ← getTraces(n.sub_1, K) ;
 8         tr_2 ← getTraces(n.sub_2, K) ;
 9         tr_out ← tr_1 ∩ tr_2 ;
10     case ∨-node
11         tr_1 ← getTraces(n.sub_1, K) ;
12         tr_2 ← getTraces(n.sub_2, K) ;
13         tr_out ← tr_1 ∪ tr_2 ;
14     case X-node
15         tr_1 ← getTraces(n.sub_1, K) ;
16         foreach s ∈ getStates(K) ∧ K, s ⊨ getSubFormula(n) do
17             foreach π ∈ tr_1 do
18                 temp ← findTraceX(s, head(π), K) ;
19                 if temp ≠ ∅ then
20                     foreach ρ ∈ temp do
21                         tr_out ← tr_out ∪ {concat(ρ, π)} ;
22     case G-node
23         tr_1 ← getTraces(n.sub_1, K) ;
24         foreach s ∈ getStates(K) ∧ K, s ⊨ getSubFormula(n) do
25             foreach π ∈ tr_1 do
26                 temp ← findTraceG(s, head(π), K, getSubFormula(n.sub_1)) ;
27                 if temp ≠ ∅ then
28                     foreach ρ ∈ temp do
29                         tr_out ← tr_out ∪ {concat(ρ, π)} ;
30                 else if s = head(π) then
31                     tr_out ← tr_out ∪ {π} ;
32     case F-node
33         tr_1 ← getTraces(n.sub_1, K) ;
34         foreach s ∈ getStates(K) ∧ K, s ⊨ getSubFormula(n) do
35             foreach π ∈ tr_1 do
36                 temp ← findTraceF(s, head(π), K) ;
37                 if temp ≠ ∅ then
38                     foreach ρ ∈ temp do
39                         tr_out ← tr_out ∪ {concat(ρ, π)} ;
40                 else if s = head(π) then
41                     tr_out ← tr_out ∪ {π} ;
42     case U-node
43         tr_2 ← getTraces(n.sub_2, K) ;
44         foreach s ∈ getStates(K) ∧ K, s ⊨ getSubFormula(n) do
45             foreach π ∈ tr_2 do
46                 temp ← findTraceG(s, head(π), K, getSubFormula(n.sub_1)) ;
47                 if temp ≠ ∅ then
48                     foreach ρ ∈ temp do
49                         tr_out ← tr_out ∪ {concat(ρ, π)} ;
50                 else if s = head(π) then
51                     tr_out ← tr_out ∪ {π} ;
52 return tr ;
```

Figure 4.8: `getTraces(`n`)`

K, which start from s_i and end at s_j. These traces fulfill formula f. Since f is a sub formula from the type $\mathbf{G}\ \alpha$, each state of these traces fulfills α. It should be noted that the last state of the traces, s_j, is omitted from the traces. This set is constructed by employing the graph search algorithm. The set returned by this function should be the set of *minimal all-state-visiting traces* with respect to formula f (see Definition 30 on the next page). The detailed algorithm of this function is presented in Section 4.6.2.

- `findTraceF(`s_i, s_j, K, f`)`
  This function takes two states s_i and s_j, model K and sub formula f as its parameters. It returns a set of traces within the model K, which start from s_i and end at s_j. It should be noted that the last state of the traces, s_j, is omitted from the traces. Sub formula f has the form $\mathbf{F}\ \alpha$. Thus, at least a state in each traces, which are collected by this function, satisfies sub formula α. Similar to the function `findTraceG()`, this function is realized by employing the graph search algorithm. Furthermore, the set returned by this function should also be the set of *minimal all-state-visiting traces* with respect to formula f. Since the algorithm of this function is almost similar to `findTraceG(`s_i, s_j, K, f`)`, we discuss further the algorithm in Section 4.6.2.

The set of execution traces returned by these sub functions fulfills the *finite set of finite visiting-all-nodes traces* property, which will be defined below.

Definition 29 (states of a trace) *the function states returns all distinct states occurring in a trace.*

Example 14 *Let* $\pi = s_0 s_1 s_3 s_2 s_2 s_4$. *Thus,* $states(\pi) = \{s_0, s_1, s_2, s_3, s_4\}$.

Definition 30 (All-state-visiting traces) *Let* Ψ *be the set of all possible traces within the model* K *and let* $states(\pi)$ *be the the set of states appearing in the trace* π. π_0 *and* π_{last} *is the first and the last state of the trace* π.

A set $\Gamma \subset \Psi$ *is defined as the set of minimal all-state-visiting traces w.r.t. model* K, *formula* f *and states* s_a *and* s_b, *if and only if,*

$$\bigcup_{\substack{\forall \pi \in \Psi : \pi \vDash f \wedge \\ \pi_0 = s_a \ \wedge \ \pi_{last} = s_b}} states(\pi) = \bigcup_{\forall \nu \in \Gamma} states(\nu) \tag{4.32}$$

and

$$\forall \nu \in \Gamma : \exists n \in \mathbb{N} : length(\nu) < n \tag{4.33}$$

are true.

The first condition in Equation 4.32 says that the union of $states(\pi)$ for each possible trace $\pi \in \Psi$, which starts from state s_a and ends at state s_b and satisfies formula f, is equal to the union of $states(\nu)$ for each $\nu \in \Gamma$. This condition also means that set Γ contains the traces, which *visit all* states that are also visited by all possible traces that satisfy formula f, start from s_a and ends at s_b. The second condition in Equation 4.33 says that the length of all traces in Γ are finite.

It should be noted that the set of all-state-visiting traces is not always unique for a given model and formula. Let us consider Example 15.

Example 15 *Let us consider a Kripke model K, which is shown in Figure 4.9. The set Γ of the model w.r.t. to formula $g = \mathbf{F}\ \beta$ and states $s_a = s_1$ and $s_b = s_5$ is $\{s_1s_2s_3s_4s_5\}$. On the other hand, one can also define $\{s_1s_2s_3s_5, s_1s_3s_4s_5\}$ as the set Γ.*

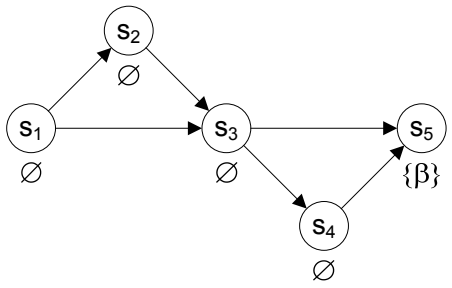

Figure 4.9: Kripke model K

4.6.2 Finding the Traces of F g_1 and G g_1 Formula

This sub section explains the usage of the strongly connected components to find the traces for **F** g_1 and **G** g_1 formulas. The algorithm to find these traces relies on the graph search algorithms to find the traces. In order to increase the efficiency, we use the strongly connected components of a directed graph.

The algorithms to find the traces for **F** g_1 and **G** g_1 are very similar. However, they differ in the fulfillment of the states of the traces. More precisely, for each trace of **F** g_1, at least one state of the trace should satisfy g_1. This condition clearly represents that g_1 will eventually hold in future. On the other hand, For each trace of **G** g_1, all of the states within this trace should satisfy g_1. Thus, this condition represents that g_1 should hold in every state of the trace.

Therefore, in order to check whether a trace satisfies **F** g_1, we can check the states of the trace. If a state in the corresponding trace exists, which satisfies g_1, then the trace fulfills **F** g_1. On the other hand, if we want to check whether a trace satisfies **G** g_1 or not, then we should check that all states within the trace satisfy g_1.

Since we use description logic-based model checking, we can easily use the instance checking service, to check whether some or all states in a set satisfy formula g_1.

Strongly connected components. According to [26], a *strongly connected component* (SCC) of a directed graph $G = (V, E)$ is a sub graph of G, whose every two nodes within the component are transitively reachable from each other. Therefore, a cycle that visits all nodes within this strongly connected component exists.

To give a better understanding of the SCC concept, we present graph $G = (V, E)$ with $V = \{a, b, c, d, e, f\}$ and $E = \{(a, b), (b, c), (c, a), (a, d), (d, e), (e, f), (f, d)\}$. This graph is shown in Figure 4.10.

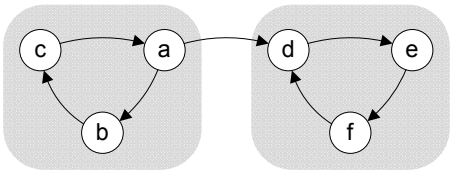

Figure 4.10: Graph G with two strongly connected components

This graph has *two* strongly connected components, which are denoted by the shaded regions. Let us consider the component, which consists of nodes a, b and c. As we can see, every node in this component is reachable from any other node within this component due to the cycle of edges. Therefore, this component is called *strongly connected component*. The strongly connected components of graph G can be represented as directed graph, whose nodes are the strongly connected components. This is shown in Figure 4.11. Thus, one can efficiently find the traces of **F** g_1 and **G** g_1 formulas.

Figure 4.11: The acyclic component graph SCC(G)

Constructing the SCC. One can compute the SCC by employing two depth-first searches on the graph G. The second run of the DFS is performed on the inverted graph, whose edges are inverted, namely G^{bot}. The algorithm can be found in [26].

Finding the Traces of F g_1 and G g_1 formula. Prior to finding the traces of **F** g_1 formula, the strongly connected components of the Kripke model should be computed. We distinguish two cases in finding the traces of this formula.

In the first case, both start and end node of the trace belong to the same strongly connected component. Therefore, we compute a single trace from the start node to the end node, which visits all nodes within this component. Such a trace may visit some nodes twice or more. In order to compute this trace, we need to compute a cycle that visits all nodes in the strongly connected component. Thus, the trace is constructed by running a sequence in the cycle from the start node to the end node, which visits every node in the cycle at least once.

The following algorithm computes the cycle:

It should be noted that the algorithm in Figure 4.12 always terminates. Since the depth first search is performed on the strongly connected component and the strategy to choose the next node is to pick the next node with the lowest visit number, all nodes in the components will be eventually visited. Therefore, it will terminate.

In the second case, the start node and the end node are contained in different strongly connected components. Therefore, a connecting trace between the two components should be constructed. This trace is constructed by performing depth first search on the acyclic graph of the SCC, whose nodes represent the strongly connected components. It starts from the component containing the start node and ends at the component containing the end node. This search returns a sequence of edges, whose endpoints belong to two different and neighboring components. In the next step, we should compute the cycle of the components, which is already presented in the first case. In particular, algorithm in Figure 4.12 is used to find such cycle. In the last step, based on the sequence of edges and the cycles of the components, we can construct the trace starting from the start node to the end node that belong to the two different strongly connected components.

```
Input: An arbitrary node a of a strongly connected component c
Output: A sequence tr_max
1  visitedNodes ← 0 ;
2  tr_max ← ∅ ;
3  i ← a ;
4  addVisit (a) ;
5  visitedNodes++ ;
6  addSequence (tr_max,a) ;
7  while visitedNodes < totalNodes (c) do
8      vst ← ∞ ;
9      foreach j ∈ nextNode(a) do
10         if visited (j) < vst then
11             vst ← visited (j) ;
12             i ← j;
13     if visited (i) == 0 then
14         visitedNodes++;
15     addVisit (i) ;
16 if i == a then
17     return tr_max ;
18 while i ≠ a do
19     vst ← ∞ ;
20     foreach j ∈ nextNode(a) do
21         if visited (j) < vst then
22             vst ← visited (j) ;
23             i ← j;
24     addVisit (i) ;
```

Figure 4.12: Algorithm to compute the *all-state-visiting* cyclic trace in a strongly connected component

4.7 Finite Set of Generated Traces

The set of traces generated by the algorithm in Figure 4.8 is a finite set. This is important, since it implies that the algorithm always terminates. In the following, we will show that the set of the generated traces is always finite.

Proposition 3 *The set of traces returned by algorithm in Figure 4.8 is always finite.*

Proof 3 Algorithm in Figure 4.8 visits each sub formula node of parse tree $\mathcal{T}_f$ starting from the root node by using recursion. In each visit, the algorithm finds the sub traces satisfying the corresponding sub formula and concatenates the traces, when necessary. To find these sub traces, this algorithm calls the sub functions `findTraceX(), findTraceG()` *and* `findTraceF()`.

Recall the sub functions `findTraceX()`, `findTraceG()` *and* `findTraceF()`. *These sub functions always return a finite set of traces, due to the minimal all-state-visiting traces property defined in Definition 30. Since the parse tree $\mathcal{T}_f$ is finite, the number of visited node is also finite and thus, the number of calls to the sub functions* `findTraceX(), findTraceG()` *and* `findTraceF()` *is also finite. Therefore, this trace generation process generates a finite set of traces.*

4.8 Example

In this section we will show an example of description logic-based model checking. Let us consider that the model K representing the workflow uses German language to describe the tasks. On the other hand, the atomic propositions appearing in the formula f use the English language.

Kripke model as Abox assertions. Let $K : \langle W, I, RL, L, AP\rangle$ be the Kripke model representing the a workflow

$$\begin{aligned} W &: \{s_1, s_2, s_3\} \\ I &: \{s_1\} \\ RL &: \{(s_1, s_2), (s_2, s_3)\} \\ L &: \{s_1 \mapsto \{bestellung\_annehmen\}, s_2 \mapsto \{rechnung\_senden\}, \\ &\quad s_3 \mapsto \{zahlung\_annehmen\}\} \end{aligned}$$

Thus, we can represent K as Abox assertions:

$$\begin{aligned} \mathcal{A}_{\mathcal{KB}} = \{&\mathsf{BestellungAnnehmen}(x_1), \mathsf{RechnungSenden}(x_2), \\ &\mathsf{ZahlungAnnehmen}(x_3), \mathit{next}(x_1, x_2), \mathit{next}(x_2, x_3)\} \end{aligned}$$

Defining the ontology of the common vocabulary. Before we define the ontology in Tbox named $\mathcal{T}_{ONT}$, we define Tbox $\mathcal{T}_{AP}$. $\mathcal{T}_{AP}$ contains the

concepts, which uniquely represents the atomic propositions in AP.

$$\begin{aligned}\mathcal{T}_{AP} \;=\; &\{\mathsf{ReceiveOrder}, \mathsf{BestellungAnnehmen},\\ &\mathsf{SendInvoice}, \mathsf{RechnungSenden},\\ &\mathsf{ReceivePayment}, \mathsf{ZahlungAnnehmen}\}\end{aligned}$$

The ontology is defined by $\mathcal{T}_{ONT}$, which is defined as follows:

$$\begin{aligned}\mathcal{T}_{ONT} \;=\; &\{\mathsf{ReceiveOrder} \equiv \mathsf{BestellungAnnehmen},\\ &\mathsf{SendInvoice} \equiv \mathsf{RechnungSenden},\\ &\mathsf{ReceivePayment} \equiv \mathsf{ZahlungAnnehmen}\}\end{aligned}$$

Translation of the formula. We specify the restricted CTL* formula, which will be translated into description logic concepts, as

$$f = \mathbf{E}\ (\mathit{receive\_order} \wedge \mathbf{F}\ (\mathit{send\_invoice} \wedge \mathbf{F}\ \mathit{receive\_payment})).$$

The parse tree of this formula is shown in Figure 4.13. The numbers appearing near the nodes are the index number of each node, which also denotes the processing order.

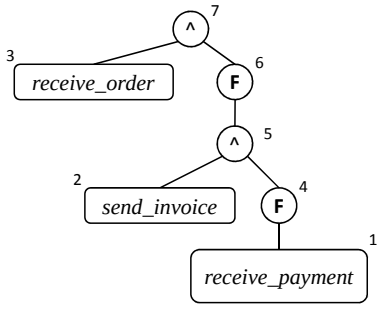

Figure 4.13: The parse tree of formula f

Starting from the root node, for each visit in a child node, an assertion is defined. Thus, the set of assertions defined by the translation process is stored in $\mathcal{T}_f$. It should be noted that the assertions defined in $\mathcal{T}_f$ are automatically generated according to Section 4.4.3. The translation schema is taken from the Table 4.7.

For example, let us consider the node #1. Since this node represents only a trivial sub formula *receive_payement*, an assertion $\mathsf{D}_1 \equiv \mathbf{K}\mathsf{ReceivePayment}$

is defined. On the other hand, the node #5 represents a more complex sub formula, which is the conjunction of two sub formulas represented by node #2 and #4. Thus, the assertion defined in $\mathcal{T}_f$ is $\mathsf{D}_5 \equiv \mathsf{D}_2 \sqcap \mathsf{D}_4$.

The whole set of concepts can be seen as follows:

$$\begin{aligned}\mathcal{T}_f \;=\; & \{\mathsf{D}_1 \equiv \mathbf{K}\mathsf{ReceivePayment}, \\ & \mathsf{D}_2 \equiv \mathbf{K}\mathsf{SendInvoice}, \mathsf{D}_3 \equiv \mathbf{K}\mathsf{ReceiveOrder}, \\ & \mathsf{D}_4 \equiv \mathsf{D}_1 \sqcup \exists \mathit{future}.\mathsf{D}_1, \mathsf{D}_5 \equiv \mathsf{D}_2 \sqcap \mathsf{D}_4, \\ & \mathsf{D}_6 \equiv \mathsf{D}_5 \sqcup \exists \mathit{future}.\mathsf{D}_5, \mathsf{D}_7 \equiv \mathsf{D}_3 \sqcap \mathsf{D}_6\}\end{aligned}$$

Performing Model Checking. Previously, we have defined $\mathcal{A}_{\mathcal{KB}}, \mathcal{T}_{AP}$ and $\mathcal{T}_f$, which are the components of the knowledge base $\mathcal{KB} : \langle \mathcal{T}_{AP} \cup \mathcal{T}_f, \mathcal{A}_{\mathcal{KB}} \rangle$.

Thus, we can simply perform model checking of

$$K, s_1 \vDash f$$

as an instance checking

$$\mathcal{KB} \vDash \mathsf{D}_7(x_1).$$

Since we only want to check, whether state s_1 satisfies formula $f = \mathbf{E}\ (\mathit{receive\_order} \wedge \mathbf{F}\ (\mathit{send\_invoice} \wedge \mathbf{F}\ \mathit{receive\_payment}))$, we do not need the algorithms defined in Section 4.6.

4.9 Summary

In this chapter, we considered the combination between the description logic formalism and the temporal logic formalism from another perspective. We provide the temporal logic reasoning on top of the description logic reasoning by *expressing* the temporal logic semantics in terms of the description logic language, which has its own formal semantics. We also defined ontology of atomic propositions that appear both in the state labels and in the temporal logic formulas. This approach facilitates the verification of the model, which uses different vocabularies used in the specification formulas. Such a case occurs when one tries to evaluate the formulas written in other vocabularies than used by the model. The next contribution of our work is that we allow the specification of CTL* formulas without the **A** operator. Since we can represent a formula as a complex concept, we could define the partial order relationships between formulas. This is very useful in the field of security research. Suppose that we specify the security policies in term of formulas.

By using the subsumption relation, we could check whether a policy encompasses another policy or not. Just like the two sides of a coin, our approach has also some limitations and disadvantages. It is rather complex to specify the CTL* formula in the description language $\mathcal{SHK}$. Furthermore, we also do not support the **A** path quantifier in the formulas, since the current description logic reasoning engine does not support epistemic operator inside a relation restriction (i.e. $\forall \mathbf{K} R.C$).

Despite of these limitations, this work would bring benefits to other fields, such as software engineering verification between heterogeneous domains and the semantic web services matchmaking [73, 109]. For example, we can use the ontology-supported model checking to perform the software verification and use the method presented in Section 4.5 to compare two temporal logic formulas, which represent the behavior of a service producer and the requirements of a service consumer, respectively. Certainly, some specifications can be represented by temporal logic formulas, which only have the existential quantifier.

Chapter 5

Refinement Patterns

5.1 Introduction

The last part of Chapter 3 mentioned the necessity of expert knowledge to refine abstract policies. In the following paragraphs, we present this argument.

Currently, most policy refinement processes are performed manually. The stakeholders and security experts discuss the stakeholders' protection plans with regard to the workflow. These plans influence the success of business objectives. As a result of such discussions, high-level workflow policies are specified as an informal text document. This document serves as a reference for deriving low-level and enforceable workflow policies. The security expert then works together with the security administrator to derive the necessary low-level workflow policies by considering the document containing the high-level policies. The goal of this step is to generate low-level policies, which satisfy the high-level policies and can be enforced by the workflow management system.

Obviously, the security expert plays an important role in this policy refinement process. He has the necessary domain-specific knowledge to capture the stakeholders' protection intent as high-level policies. On the other hand, the security expert also has profound knowledge in the security engineering area, which is required to derive the enforceable security policies.

The main concept of refinement patterns has been published in [78]. However, we extend the concept and present it in this chapter.

Motivation. Our goal is to capture expert knowledge about solving a particular kind of problem, and in this case, the problem of refining workflow policies. However, adopting knowledge from experts is not a trivial task.

Some experts gain their knowledge through experience in solving the problems. Thus, we need a well-proven method to capture the expert knowledge in problem solving. Furthermore, we need that the expert knowledge to be machine interpretable. Therefore, a formal representation of the captured knowledge is required. Based on these requirements, we will define policy refinement patterns adapted from pattern paradigms and represent them in formal notation.

Outline. Since we are adapting the pattern paradigm, we will explain it in the next section. Section 5.3 discusses related work, which deals with the adaptation of the pattern paradigm and the pattern formalization. Section 5.4 describes our definition of policy refinement patterns. In Section 5.5, we define the different classifications of the policy refinement patterns. Section 5.6 defines the formal representation of the policy refinement patterns database. Section 5.7 discusses an approach to mine the policy refinement patterns. Section 5.8 presents the example of a policy refinement pattern. Section 5.9 concludes this chapter with a summary.

5.2 Background

To design a building, architects consider many non-technical and human-related factors, for example, social or psychology factors. This is necessary, due to the fact that people live and interact in the building and have their own needs. Therefore, architects design constructions, which fulfill the needs emerging from these factors.

Christopher Alexander proposed the *pattern* method in his work [4], which structurally captures expert knowledge. Each pattern has three general parts, a **context**, a **problem** and a **solution**. The context describes the environment, in which the pattern applies. It covers both temporal and spatial aspects of the environment that represent a scenario. Furthermore, *forces*[1] exist within the context that should be resolved. The forces characterize the problem in the context. Finally, the solution proposes a configuration or a design which resolves the existing forces in the context. Collected patterns do not exist independently; they have one or more relationships with other patterns. These relationships reflect the conflicts, compatibilities and dependencies between patterns applied within a context.

[1] In our work, we interpret forces as requirements and problems.

5.3 Related Work

This section presents several approaches that are related to ours.

Pattern Approach in Software Engineering. The most known adaptation of pattern paradigms is the *design patterns*. The goal of the design pattern approach is to capture the best-practice solutions to recurring problems in object-oriented software design. This approach is made famous by the book of Gamma et al. [42].

The history of design patterns began with the experiment conducted by Kent Beck and Ward Cunningham [12]. They used the *pattern language* approach to write down the reference guide to designing window-based user interface for Smalltalk as a collection of patterns. They presented the patterns to a team of application specialists, who only have a basic knowledge of Smalltalk's user interface mechanisms. Apparently, the team was able to specify very reasonable interfaces in Smalltalk language after one day practice. This success initiated the adoption of the pattern method to capture expert knowledge about designing software architecture, since design patterns facilitate the reuse of the expert knowledge in building object-oriented software.

The structure of a pattern defined by Christopher Alexander has only the essential parts, namely *problem*, *context* and *solution* [4]. However, the structure of a design pattern, which is adopted from the Alexander's definition of pattern, consists of four parts, *pattern name*, *context and problem*, *solution* and *consequences* [42]. These parts are written in informal texts. Additionally, the solution is also described by using UML diagrams. Thus, the design pattern has an informal or semi-formal representation.

Nevertheless, the informal or semi-formal representation of the design patterns causes the ambiguous interpretation of design patterns. Garlan et al. stated that some problems arise during the reuse of design patterns, since the developers may ambiguously identify the design patterns due to the informal representation [43]. Therefore, wrong patterns may be applied by the developers.

Pattern Approach in Security Engineering. Schumacher presented in [91] a security engineering process, which is based on the pattern paradigm. He defined *security patterns*, which aim at capturing the expert knowledge in security improvement. He attempted to formalize *security patterns* by using Frame Logic. The formalization of the security patterns enables the automated search of security patterns from the pattern catalogue. It helps the

security administrators to find the right security patterns from a large collection of patterns efficiently. The structure of a security pattern consists of name, context, problem and solution, which is similar to the structure defined by Christopher Alexander. Additionally, each security pattern can have a relationship to other patterns. There are several types of relationships between security patterns., they are *refines, uses* and *conflicts*. Each description of the context, problem and solution contains a set of terminologies.

It should be noted that, for our purpose, frame logic is not expressive enough to represent the refinement patterns. Although description logic originates from frame logic [7], frame logic lacks several extensions, which are needed to perform the description logic-based model checking. This is necessary since we want to perform automated pattern matching. These extensions are the addition of the epistemic operator **K** and the hierarchical definition of relationships between the concepts.

Pattern-based Code Generation The pattern approach was developed in order to communicate expert knowledge (best-practice knowledge) between humans. However, we also aim at generating policies from patterns. We will investigate existing approaches that aim for generating code from design patterns.

Budinsky et al. presented an approach to automatic code generation from design patterns [20]. To achieve their goal, they augmented each design pattern with a template of code, which corresponds to the solution of the design pattern. The description of each design pattern is represented in hypertext format. Thus, the user interacts with the pattern catalogue through the web-browser user interface.

To generate the code from the design patterns, the user should select one. Through the web-browser interface, the user can read the pattern description and enter additional information, which serves as the parameter values for the code template. Finally, the user can generate the code from the template, which is filled by the parameter values entered before.

5.4 The Structure of a Refinement Pattern

The structure of a refinement pattern consists of three parts, namely, **context**, **problem** and **solution**. Thus, it is similar to the basic structure of a pattern. These parts are defined in the next subsections.

5.4.1 The Context of a Refinement Pattern

The *context* of a pattern points out the temporal and spatial location, in which the pattern can be applied. In our case, the refinement pattern should be applied to *a certain execution trace* within the workflow. The experts know that the execution trace fulfills the desired abstract security property, which is subject to policy refinement. Therefore, the context of this pattern characterizes the execution trace of the workflow, which is described by formula ϕ_{ctx}. Example 16 shows the context of a pattern.

Example 16 *Let us consider formula* $\phi_{ctx} = E(apply\_loan\_application \wedge$ **F** $approve\_loan\_application)$*. Informally, this formula satisfies any sequence that starts with "apply loan application" task and eventually succeeded by "approve loan application" task. Therefore, the context of the pattern characterizes a fragment of task execution sequence, which starts with "apply loan application" and eventually executes "approve loan application".*

5.4.2 The Problem of a Refinement Pattern

The *problem* of a pattern describes the unfulfilled goals or requirements within the context, which should be solved. In our case, the problem described by the refinement pattern represents the abstract property, which will be refined by the refinement pattern. However, the desired abstract security property is represented as the state labels along the execution trace. Therefore, the problem of the refinement pattern can be represented as an atomic proposition *prb*. Let us consider Example 17.

Example 17 *prb = secure_loan_application_process informally says that all states within the fragment of the context has the label "secure loan application process". It means also that the security property called "secure loan application process" holds within the fragment of the task execution sequence.*

5.4.3 The Solution of a Refinement Pattern

The *solution* of a pattern describes the constructive design proposed by experts, which should solve the problem mentioned before. In other words, the solution of a pattern should extend the existing or generate a new construction. This construction is the *best-practice* solution. It means that this solution has been applied in several similar problems within similar contexts. Furthermore, the experts conclude that this solution can address these problems. Therefore, this solution has been proven to be effective to the problem within the context, which are described by the pattern.

In our case, the solution of a refinement pattern has two purposes. First, it describes the *addition* of new labels to the states within the fragment of task execution sequence, which is matched by the problem and the context. These addition of new labels are proposed by the workflow security experts. The second purpose of the solution of the refinement patterns is to *generate* an enforceable policy, which can be interpreted by machine. In our work, we consider Extensible Access Control Mark-up Language (XACML) as our enforceable policy. The enforceable policy shall address the security problem within the context, which are described by the corresponding pattern.

The security experts have applied this solution in recurring problems and found out that this solution, either the addition of state labels or the generation of enforceable policy, is effective in addressing the corresponding problems. This means that the validity of the solution is *proved* by the empirical facts. In general, we conclude that a best-practice empirical proof is sufficient to claim that a solution of the refinement pattern is effective in addressing the corresponding security problem within the fragment of task execution sequence. However, although not necessary, we will add a mathematical proof, which verify that that the solution solves the problem within the context.

Abstract Solution

The abstract solution describes the security properties, which should be added as a new set of state labels of the Kripke model. According to the experts, these security properties should satisfy the security property described in the problem of the pattern. Since the less abstract desired properties are represented in the Kripke model as state labels along the execution traces matched by the context, we can represent the solution only as a set of atomic propositions. Let us consider Example 18.

Example 18 *Let sln be a set consisting of atomic propositions, which is defined as follows:* $sln = \{prevent\_fictive\_loan\_application, prevent\_unauthorized\_access\}$. *Informally, the sln propose a set of new state labels, which should be added in every states of the traces, which are matched during the pattern matching.*

Concrete Solution

The concrete solution describes the template of the enforceable policy, from which an enforceable policy will be generated, if this solution is instantiated. We use the XACML standard to specify the enforceable policies. Thus, the

solution of the pattern describes the template of the XACML policy, which uses the XML standard.

Every template has several *parameters*, which will be filled by *task names*. The task names are taken from the state labels of the execution traces, which match with the context and the problem of the corresponding pattern. In this work, only task names determine the parameters of the generated XACML policies. To give a better understanding of the concrete solution, an example of the solution for separation of duty policy presented as follows.

Example Solution for Separation of Duty. The "separation of duty" concept was first introduced by Saltzer and Schoeder [83]. They introduced this concept as one of the eight design principles for the protection of the information in IT-system.

The motivation of this concept is to avoid the accidental execution of two sensitive tasks by *a single user* that can cause harm to the stakeholders (or to everyone). Obviously, allowing the execution of two sensitive tasks by a single user may pose more risks to the stakeholders than separating the permission to execute the two sensitive tasks by two users. Another motivation of this concept is to split the responsibility and the authority in executing the two sensitive tasks into two users. Thus, it discourages potential fraud.

In the original concept of separation of duty, the order of the tasks' execution, which can cause harm, is not considered. If any of the two tasks is executed by a user, the execution of the other task by the user should be prohibited. In our work, however, we consider the sequence of the tasks' execution, which may cause harm. For example, the subsequent execution of *createLoanAppl* and *approveLoanAppl* is considered as harmful, whereas the subsequent execution of *approveLoanAppl* and *createLoanAppl* is not harmful. Therefore, the corresponding separation of duty policy generated by the solution of the pattern only prevents the subsequent execution of *createLoanAppl*, which is followed by *approveLoanAppl*.

To realize separation of duty mechanism with XACML policy, we adapt the work by Jason Crampton [27]. The basic idea of his work is to create the *triggering* policy and to maintain a dynamic blacklist written in XACML policies. If a user executes one of the sensitive tasks, which should be separated, then the *triggering* policy dynamically creates an XACML prohibition policy. This policy denies the execution of the other sensitive tasks by the same user. The detailed discussion of the Crampton's approach, which uses XACML policy to enforce the separation of duty mechanism can be found in [27].

In this case, the concrete solution shall generate the triggering XACML

```
<PolicySet ...  PolicySetId="Permission:Set" ...  >
  <Policy ...  PolicyId="permission:po:create" ...  >
    <Target>
      TaskID = "Parameter #1"
    </Target>
    <Obligations ...  >
      <Obligation FulfillOn="Permit" ...  >
        addDenyPermissionToSubjectBlacklist
        ("Parameter #2")
      </Obligation>
    </Obligations>
  </Policy>
  .
  .
  .
</PolicySet>
```

Figure 5.1: The template of an enforceable XACML policy for separation of duty

policy, which prevents the subsequent execution of two sensitive tasks. Figure 5.1 shows the template of the triggering XACML policy. To generate the XACML policy from the template, the Parameter #1 and Parameter #2 should be substituted by the names of the first and the second sensitive tasks, respectively.

5.5 Classification of Policy Refinement Patterns

In this section, we will define the classification of policy refinement patterns, which will be used in policy decomposition. The purpose of this classification is to guide the pattern administrator to identify the dependencies between the patterns.

Each decomposition of a policy by a refinement pattern introduces several new policies. If these policies could be further decomposed, other patterns would decompose these policies. Thus, we see that the previous pattern should be *first* applied, before the other patterns are applied.

In the following, we present the classification of the refinement patterns:

- Abstract refinement pattern
  The abstract refinement pattern is intended to decompose specified policy (N_{spec}) and abstract policy (N_{abs}) into several abstract policies. Thus, this kind of patterns propose abstract policies as their solutions.

- Concrete refinement pattern
  The mechanism refinement pattern decomposes abstract policy (N_{abs}) into several concrete policies (N_{con}). The concrete policies state the application of security mechanisms or authorization constraints, which can fulfill the abstract policy.

- Enforceable refinement pattern
  This pattern instantiates a template of enforceable policy specification, which is derived from a concrete policy. This pattern proposes a template of machine-interpretable policy specification. In our case, the pattern generates XACML policies from templates.

The following table shows the overview of categorization of the policy refinement patterns:

pattern type	type of the refined policy	type of the generated policies
abstract refinement pattern	specified or abstract policy	abstract policy
concrete refinement pattern	abstract policy	concrete policy
enforceable refinement pattern	concrete policy	enforceable policy

Table 5.1: Classification of the workflow policy refinement patterns

5.6 Formal Representation of the Database

The formal representation of the pattern database allows the machine interpretation of policy refinement patterns. This is important, since we want to perform an automatic pattern matching. In the pattern matching, the applicability of each pattern is checked. If the pattern is applicable to the Kripke model representing the workflow behavior and workflow policies, then the pattern matching has found a set of execution traces, which match the description of the context and the problem. Thus, the solution can be applied in order to refine the abstract workflow policies.

As previously mentioned in Section 5.4, the context of a pattern is represented as a temporal logic formula. On the other hand, the problem and the solution of the pattern is represented as an atomic proposition and a set of atomic propositions, respectively. Since we already defined the restricted CTL* semantics on top of the description logic semantics $\mathcal{SHK}$, we can represent the context, the problem and the solution of the pattern in terms of description logic concepts.

A refinement pattern database is a description logic-based knowledge base, which consists of several Tboxes. These Tboxes are represented by description logic language $\mathcal{SHK}$, which was introduced in Section 4. Let $\mathcal{T}_{KB}$ be the Tbox representing all Tboxes. The Tbox is defined as follows:

$$\mathcal{T}_{KB} := \underbrace{\mathcal{T}_{ctx}}_{\mathcal{T}_f} \cup \underbrace{\mathcal{T}_{taskname} \cup \mathcal{T}_{securityproperty}}_{\mathcal{T}_{AP}} \cup \mathcal{T}_{Pattern} \cup \mathcal{T}_{ONT}$$

$\mathcal{T}_{ctx}$ is the Tbox that describes the context of each pattern. It contains the ontologies that represent the translated temporal logic formulas of the context of the patterns (see Section 4.4.3). Therefore, $\mathcal{T}_{ctx}$ is from the type of $\mathcal{T}_f$. On the other hand, we do not see any Tboxes, which represent problems and solutions, as a part of the $\mathcal{T}_{KB}$. This is due to the fact that *prb* and *sln* are elements of AP, and AP is already represented by $\mathcal{T}_{securityproperty} \cup \mathcal{T}_{taskname}$.

$\mathcal{T}_{taskname}$ and $\mathcal{T}_{securityproperty}$ are from the type of $\mathcal{T}_{AP}$, which define the atomic propositions in terms of concepts (see Section 4.4.4). $\mathcal{T}_{taskname}$ is constructed by defining a unique concept name for each atomic proposition representing the task name in $AP_{taskname}$. $\mathcal{T}_{securityproperty}$ is constructed by defining a unique concept name for each atomic proposition representing the security property in $AP_{securityproperty}$. Additionally, $\mathcal{T}_{ONT}$ represents the semantic relations between different concepts of $\mathcal{T}_{securityproperty}$ and $\mathcal{T}_{taskname}$.

The $\mathcal{T}_{Pattern}$ Tbox contains the definition of concepts, which formally define the *abstract refinement pattern*, *concrete refinement pattern* and *enforceable refinement pattern*. Formally, $\mathcal{T}_{Pattern} = \mathcal{T}_{ARP} \cup \mathcal{T}_{CRP} \cup \mathcal{T}_{ERP}$. These concepts are formally defined as follows:

Definition 31 *An abstract refinement pattern and a concrete refinement pattern are represented by the concepts* AbsRefPatt *and* ConRefPatt*, respectively. These are defined as follows:*

$$\mathsf{AbsRefPatt} \equiv \mathsf{ConRefPatt} \equiv \exists \mathsf{hasContext}.\mathsf{D}_{ctx} \sqcap \exists \mathsf{hasProblem}.\mathsf{D}_{prb} \sqcap \exists \mathsf{hasSolution}.\mathsf{D}_{sln} \ldots \exists \mathsf{hasSolution}.\mathsf{D}_{sln}.$$

$\mathsf{D}_{ctx} \in \mathcal{T}_{ctx}$ *represents temporal logic formula* ϕ_{ctx}*. The transformation of the temporal formula into the description logic concept is defined in Section 4.4.3.* D_{prb} *represents atomic proposition prb and* D_{sln} *represents an atomic proposition within sln. Since this concept may have several relationships with different concepts representing the solutions, each pattern has a set of solutions.*

Let us consider the next example, which illustrates the abstract refinement pattern:

Example 19 *The concept* $\mathsf{PreventFictiveLoanApplicationPattern} \equiv \exists \mathsf{hasContext}.D_1 \sqcap \exists \mathsf{hasProblem}.\mathsf{PreventFictiveLoanApplication} \sqcap \exists \mathsf{hasSolution}.\mathsf{ApplySoD}$ *represents the pattern called "Prevent Fictive Loan Application". Furthermore, the concept has a relationships with concepts* D_1*, which is defined in the* $\mathcal{T}_{ctx}$*. This Tbox is defined as:* $\mathcal{T}_{ctx} = \{D_1 \equiv (\mathbf{K}\mathsf{SubmitLoanApplication} \sqcap (\mathbf{K}\mathsf{ApproveLoanApplication} \sqcup \exists \mathsf{future}.\mathsf{ApproveLoanApplication}))\}$*.*

Definition 32 *An enforceable refinement pattern is represented by the concept*

$$\mathsf{EnfRefPatt} \equiv \exists \mathsf{hasContext}.\mathsf{D}_{ctx} \sqcap \exists \mathsf{hasProblem}.\mathsf{D}_{prb} \sqcap \exists \mathsf{hasSolution}.\mathsf{XACMLTemplate}.$$

$\mathsf{D}_{ctx} \in \mathcal{T}_{ctx}$ *represents the temporal formula* ϕ_{ctx}*.* D_{prb} *represents atomic proposition prb. The solution of this pattern (sln) is the template of a security policy, which is machine interpretable. Since we take XACML as our example, the concept* XACMLTemplate *represents the template of the XACML policy, which realizes the desired authorization constraint. The concept* XACMLTemplate *is an OWL concept, which can store the template.*

Example 20 shows an enforceable refinement pattern.

Example 20 *The concept* $\mathsf{ApplySoDXacmlPattern} \equiv \exists \mathsf{hasContext}.D_1 \sqcap \exists \mathsf{hasProblem}.D_4 \sqcap \exists \mathsf{hasSolution}.\mathsf{XacmlSoD}$ *represents the pattern called "apply separation of duty with XACML policy". Additionally, this concept has relationships with concepts* D_1 *and* D_4*. The concept* D_1 *is already defined in Example 19 and* D_4 *is defined in* $\mathcal{T}_{prb} = \{D_4 \equiv \mathsf{ApplySoD} \sqcap \exists \mathbf{K}\mathsf{next}.(\mathsf{ApplySoD} \sqcup D_4)\}$*. Since we use OWL to represent the Tbox, a concept may contain a string. Therefore,* XacmlSoD*, which is the solution, is represented in OWL as follows:*

```
<owl:Class rdf:ID="XacmlSoD">
<rdfs:comment rdf:datatype="http://www.w3.org/2001/
XMLSchema#string">&lt;PolicySet PolicySetId=
"Permission:Set" &gt;&lt;Policy PolicyId=
"permission:po:create" &gt;&lt;Target&gt;TaskID =
"Param#1"&lt;/Target&gt;&lt;Obligations&gt;&lt;
Obligation FulfillOn="Permit"&gt;
addDenyPermissionToSubjectBlacklist ("Param#2")&lt;
/Obligation&gt; &lt;/Obligations&gt; &lt;/Policy&gt;
&lt;/PolicySet&gt;</rdfs:comment>
</owl:Class>
```

From the definitions above, all refinement patterns are represented by the concepts that have three kinds of relationships. These relationships are *hasContext*, *hasProblem* and *hasSolution*.

5.7 Mining Policy Refinement Patterns

Several approaches for pattern mining are identified by Kerth and Cunningham [55]. They are:

- Sociological approach:
  In this approach, several experts are asked how they solve particular problems and why this was a good solution. Currently, interviews and questionnaire are used to gather the knowledge from the experts. Furthermore, an approach called *shepherding* is also used to control the quality of the gathered patterns. In this approach [48], one writes his own experience in solving a problem as a pattern. Thus, he sends the pattern to a supervisor, who reviews the pattern. This phase is known as *shepherding*. The supervisor, which is also known as the shepherd, has the job to help the pattern author improve the quality of his pattern. Thus, the shepherd should be an experienced pattern author and has the necessary background knowledge. In the end, the pattern is introduced in a workshop and will be evaluated.

- Introspective approach:
  In this approach, one analyzes the solutions, which *he has made*, and tries to identify his solutions that successfully address the problem. However, this approach only captures patterns, which are only limited to personal experiences.

- Artifactual approach:
  In this approach, one analyzes the solutions, which *others have made*, and tries to identify the commonalities or similarities in their solutions. Thus, these findings are written as a pattern, which generally describes the solution.

In the scope of a master thesis [1], we present a novel approach to capture the expert knowledge in refining workflow policies as refinement patterns. Inspired by the Wikipedia, which gather an enormous knowledge through collaborative works of several authors, we use the semantic wiki system [98, 93, 87] to acquire workflow refinement patterns. As we can see from the previous categories of pattern mining approaches, we classify the approach presented in the master thesis as *sociological approach*. This approach aims for the following goals:

- A more effective way to gather expert knowledge by motivating all experts to proactively collaborate together in writing down their knowledge,
- The knowledge written by the experts can be automatically transformed into ontologies, which is machine interpretable

Therefore, this approach offers some advantages compared to the traditional sociological approach. Since the work is collaboratively executed by the experts around the world, who use the semantic wiki platform, we could expect these advantages:

- The pattern database contains the actual and up-to-date patterns. Since the patterns authoring and evaluation process is performed online, thus all experts and pattern authors can instantly review and improve the patterns.
- More involvement and participation of the experts in writing down their knowledge as patterns is expected. This is possible, because the on-line semantic wiki platform facilitates the collaborative work around the world.

Nevertheless, we also identify the following disadvantages of a wiki system:

- The administration of the semantic wiki system is needed to prevent pranks. Such problems are widely known in the wikipedia platform that leads to the degradation of its quality. However, this problem can be addressed by recruiting administrators and experts, who supervise the system.

- The experts should write their knowledge in a rigid structure, which offers less flexibility in writing the knowledge. However, the current structure should be sufficient for our approach.

As stated before, we use a semantic wiki tool to capture the expert knowledge from humans. This tool has the similar user interface to the ordinary wiki engine. It has the web interface, which experts can interact with. Through this interface, the users document their knowledge as *articles*. Like any other wiki-based knowledge management systems, semantic wiki also documents expert knowledge as articles. The experts can create, edit and revise the articles. Furthermore, the experts can also discuss, make correction and keep several versions of the articles. An article commonly contains a terminology and its explanation. Therefore, the semantic wiki tool translate the article into a *concept*. Furthermore, the article can also have *links* to any other articles. These links are used to create the *semantic relationships* between the concepts. Thus, the semantic wiki tool has the ability to automatically transform the articles and their links into ontologies, which comprise of concepts and relationships.

In our work, the experts write the articles that describe workflow tasks, security properties and refinement patterns. It means that each time the experts write or correct the articles, the semantic wiki tool can *automatically generate* the corresponding ontologies. Therefore, the semantic wiki also plays a significant role in bridging the knowledge transfer between human and machine.

However, we also require that the semantic wiki tool should also be able to document the temporal logic formulas, which describe the context and the problem of the patterns. Therefore, we extended the semantic wiki tool by adding a simple parser of temporal logic formulas and a formula to concept translator module [1].

Usually, the experts should first store the articles describing the terminologies of workflow tasks and security properties. Thus, the experts can create articles about refinement patterns. Each creation of article about the refinement patterns, a template is used. This template contains three parts, which are the context, the problem and the solution of the pattern. Each of this part has two type of descriptions, namely, informal and formal descriptions. The informal description is the textual description of the context, the problem or the solution. On the other hand, in the formal description, the experts can specify the temporal formulas or the define the set of security properties. The former is needed to describe the context and problem, whereas the latter is needed to describe the solution. By using this approach, we can automatically derive the necessary ontologies, which describe the re-

finement patterns, from the semantic wiki tool.

The source of the mining process can range from the BSI IT baseline protection catalog [19] to ISO 27002:2005 [53]. However, we will present an approach to extract the expert knowledge from the BSI IT baseline protection catalog.

Importing BSI IT Baseline protection catalog. The IT baseline protection catalog is a document that describes both the identification of the threats in IT environment and the safeguards addressing the threats. It is developed by Federal Office for Information Security *(Bundesamt f́ur Sicherheit in der Informationstechnik)*. Hence, it is also known as BSI IT baseline protection catalog.

The catalog is divided into three sub catalogs: *Modules*, *Threats* and *Safeguards*. The modules serves as the main entry point for identifying and addressing the threats in IT environment. It means that the administrator should first read the modules, in order to identify the classification of his asset, which is exposed to the security risk. Further, each entry in the modules is linked both to some entries in the threat and to some entries in the safeguards. The links between modules and threats indicate that the threats may pertain to the components, to which they are related. On the other hand, the links between modules and safeguards determine the security measures, which address the threats related to the considered components.

The aim of the IT baseline protection is to facilitate the security administrator in composing security concepts and mechanisms, which address the security risks. The composed security concepts and mechanisms are based on the standard security measures proposed by the IT baseline protection catalogs. Since these security concepts are meant to be interpreted and realized by the administrator, they cannot be directly enforced and realized in the machine. Therefore, the refinement patterns derived from the IT baseline protection are from the type of the abstract refinement patterns.

Deriving refinement patterns from the IT baseline protection is relatively simple, since the categorization of the IT baseline protection sub catalogs almost reflects the sub parts of a refinement pattern. The modules, threats and safeguards are related to the context, problem and solution of the pattern, respectively. However, there are some differences. The refinement pattern database aggregates each context, problem and solution into a triplet, which is called pattern. Thus, this database contains a collection of patterns. On the other hand, the IT baseline protection does not contain such aggregation. Therefore, the experts, who read the IT-baseline protection, should construct the triplet $(context, problem, solution)$ aggregation during

the derivation process.

Another obstacle in deriving refinement patterns from IT baseline protection lies in the description of the entries in the modules. The entries in the modules contain only the static description of the object (asset). In contrast, the context of the pattern describes the fragment of the organizational process. Hence, it describes the activities that use the considered object (asset). Thus, deriving the context from the modules requires the construction of the scenario, which describes the activities using the considered object (asset).

In the following, we explain the two steps, which are required to derive a refinement pattern from the IT baseline protection. These processes are repeated until each threat in the corresponding modules is aggregated together with the corresponding measure in a refinement pattern. These steps are explained as follows:

Step 1: Constructing the context and the problem. This step begins with the analysis of the entries in the modules. As already discussed above, deriving context from the modules requires the construction of the scenario. Each scenario describes a threat occurring within the considered modules. Since each entry of the modules contains some links to the threats, several scenarios describing the threats should be constructed. Based on these scenarios, a pair of the context description and the problem description of a refinement pattern is then constructed.

Step 2: Adding the solution. The entry of the modules also contains the links to the corresponding safeguards, which can address the threats. For each threat occurring in the modules, the appropriate security measures are determined by the links. Thus, the solution of the pattern is derived from the security measures. It should be noted that the solution may contain several security measures. Furthermore, the security measures described in the safeguards of IT baseline protection does not contain any machine-enforceable security policies. Thus, we can only derive abstract refinement pattern from the IT baseline protection.

The following example shows a refinement pattern derived from the IT baseline protection.

Example 21 *Let us consider the module "Servers under Windows NT" (B 3.103) in IT baseline protection catalog. This module describes the operation of servers, which run on Windows NT operating system. From this module, we should derive the context of the refinement patterns. Since this module does not describe the scenario as a sequence of events, we should*

therefore construct the sequence of events by ourselves. We construct the sequence as follows: $\phi_{ctx} = E(Admin\_Login \wedge \mathbf{F}\ (Admin\_Operates\_Server \wedge \mathbf{F}\ Admin\_Logout))$. *It means that the administrator logs in into the server and operates the servers. Finally, he will then log out from the server.*

The module has several links to some threats. Since we only want to extract only a refinement pattern, we consider the threat, which is called "Computer Viruses" (T 5.23). It means that during the whole process of operating the server, there is a threat, which is caused by computer viruses. Thus, we identify this as a problem of the pattern describe the problem as follows: $prb = computer\_virus$.

Furthermore, this module also points out the safeguards that address this problem. These safeguards are "Disable automatic CD-ROM recognition" (S 4.57), "protect registry" (S 4.75) and "safe-guarding boot up procedure" (S 4.49). Thus, the solution is defined as follows: $sln = \{protect\_registry, disable\_automatic\_cdrom\_recognition, protect\_registry\}$ *The prb, ctx and sln are then grouped together as a triplet of the refinement pattern.*

5.8 Example

In this section we will present an example of the refinement pattern both in informal and in formal representation. We take the refinement pattern from the type of *plan refinement pattern* as the example. This pattern is derived from the paper by Schaad and Moffett [86].

Introduction. The name of this pattern is "Protect Loan Application Process". The purpose of this pattern is to derive less abstract workflow policies, which fulfill the stakeholders' protection intent with regard to a loan application process.

Context. The context of this pattern is the fragment of the tasks' execution sequence with regard to the loan application process. Typically, the fragment starts with the task named "loan application submission" and ends with the task named "send e-money payment to client". Formally, this fragment is represented as $\phi_{ctx} = E(loan\_appl\_submission \wedge \mathbf{F}\ send\_payment)$.

Problem. The problem found in the above mentioned context is the abstract desired property called "secure the loan application process". This problem is represented as $prb = protect\_loan\_appl\_process$.

Solution. The solution to this problem is to propose policies that:

- prevent fictive loan,
- prevent unauthorized access to the sensitive financial tasks in loan application process.

This solution is represented as $sln = \{prevent\_fictive\_loan\_application, prevent\_unauthorized\_access\}$.

Description logic representation. The set of concepts defining this pattern is defined as follows. It should be noted that the abbreviation "plap" stands for "protect loan application process".

$$
\begin{aligned}
\{\mathsf{Pat}_{plap} &\equiv \exists \mathit{hasContext}.\mathsf{D}_{ctx\_plap} \sqcap \exists \mathit{hasProblem}.\mathsf{D}_{prb\_plap} \sqcap \\
&\quad \exists \mathit{hasSolution}.\mathsf{D}_{sln\_plap\_1} \sqcap \exists \mathit{hasSolution}.\mathsf{D}_{sln\_plap\_2};
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{D}_{ctx\_plap} &\equiv \mathsf{D}_2 \sqcap \mathsf{D}_3; \\
\mathsf{D}_2 &\equiv \mathbf{K}\mathsf{LoanApplSubmission}; \\
\mathsf{D}_3 &\equiv \mathsf{D}_1 \sqcap \exists \mathit{future}.\mathsf{D}_1; \\
\mathsf{D}_1 &\equiv \mathbf{K}\mathsf{SendPayment}; \\
\mathsf{D}_{prb\_plap} &= \mathsf{ProtectLoanApplProcess};
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{D}_{sln\_plap\_1} &\equiv \mathsf{PreventFictiveLoan}; \\
\mathsf{D}_{sln\_plap\_2} &\equiv \mathsf{PreventUnauthroizedAccessInLoanApplProcess}\}
\end{aligned}
$$

The concept Pat_{plap} represents the pattern itself. This concept has four relationships. They are a relationship to a context (D_{ctx_plap}), a relationship to a problem (D_{prb_plap}) and two relationships to two solutions ($\mathsf{D}_{sln\_plap\_1}$ and $\mathsf{D}_{sln\_plap\_2}$). The original representation of the context is temporal logic formula. Since we need the description logic representation of the context, we translate it to concepts according to the translation rules defined in Section 4.4.3.

5.9 Summary

In this chapter we defined the structure, the representation and the classification of policy refinement patterns. These definitions shall guide the pattern authors in writing the patterns. The pattern paradigm is an effective method

to structurally document the expert knowledge. Therefore, it has been used several fields of computer science [18, 42, 91].

Furthermore, this section also briefly discussed a novel approach in pattern mining, which was realized within the scope of a master thesis [1]. In this approach, we took the advantage of wiki system and semantic web technology (i) to effectively capture the expert knowledge and (ii) to write the expert knowledge in machine interpretable representation. As a result, it can be immediately used by the policy refinement process.

Nevertheless, our approach still has some limitations. The refinement patterns can only capture the *known* security problem with the corresponding solution. Thus, a new identified security problem cannot be addressed by the refinement patterns. Another limitation of the refinement patterns is that the solutions of abstract refinement patterns cannot propose state labels with parameters.

In the next chapter, we will discuss the policy refinement process, which uses the building blocks presented in Chapter 4 and Chapter 5 to generate the policy refinement tree defined in Chapter 3.

Chapter 6

Automated Policy Refinement Process

6.1 Introduction

This chapter presents the automated policy refinement process, which is built upon the building blocks. These building blocks are defined in the previous chapters. They are:

- formalization of the workflow behavior and the high-level policy specification,
- the definition of the policy refinement tree,
- the definition of policy refinement patterns and
- the definition of description logic-based model checking.

The first building block deals with the the formal representation of the workflow behavior and the high-level policy specification in a Kripke model. The automated refinement process is performed by the machine, which should be able to interpret and to analyze the workflow behavior and the high-level policy. Therefore, the workflow behavior and the high-level policy should be formally represented.

Although we can specify the workflow policies as state labels, as defined in Section 2.4, this method is not expressive enough to represent the *dependencies* between a policy and its sub policies. The sub policies and their dependencies are generated by the refinement process. Therefore, we propose the second building block, which is the policy refinement tree. With the policy refinement tree, we can specify the *dependencies* between a policy

and its sub policies. These dependencies are categorized into *OR* and *AND* connections. The former means that according to the refinement process, the policy is fulfilled, if and only if, all of its sub policies are fulfilled. The latter means that according to the refinement process, the policy is fulfilled, if and only if, one of its sub policies is fulfilled. Thus, we can also conclude that the refinement process results in a policy refinement tree.

The third building block defines the policy refinement patterns. Previously in the end of Chapter 3, we argued that the refinement of policies requires expert knowledge. In this building block, we use pattern paradigm to capture the expert knowledge in refining workflow policies. Furthermore, we also define the formal representation of policy refinement patterns. Since the machine should *also* be able to interpret the expert knowledge, formalization of the policy refinement patterns is required.

The fourth building block defines a novel approach in combining description logic formalism with temporal logic formalism. By combining both formalism, we establish the description logic-based model checking, which facilitates the automated pattern matching. The pattern matching is conducted during the refinement process and aims for finding the fragment of the execution trace in the workflow that matches the description of the refinement patterns. This particular fragment of the execution trace represents the abstract workflow policy, which is subject to the refinement process.

The concept of the automated policy refinement process has already been published in [79] and [81].

Motivation. The high-level policies specified by the stakeholders are still too abstract to be enforced by the machine. By consulting the refinement patterns database, the policy refinement process should iteratively decompose each abstract policy into several less abstract policies. Intuitively, one can see the hierarchical composition relationships between the policy and its sub-policies as a policy refinement tree, which is defined in Chapter 3. The policy refinement process stops if all leaves of the policy refinement tree represent concrete policies.

Outline. This chapter begins with the discussion of the related work in the policy refinement process. Subsequently, three important steps in the policy refinement process are discussed. Section 6.3 discusses the pattern matching step, which should find the abstract policies that are subject to the refinement. Section 6.4 discusses the instantiation process of pattern solutions. Section 6.5 presents the algorithm of the refinement process. This section is then followed by an example in Section 6.6. Discussion about the

policy refinement algorithm can be found in 6.7. Section 6.8 summarizes this chapter.

6.2 Related Work

Workflow constraint generation. In the field of workflow constraint specification process, our work is similar to the work of Neumann and Strembeck [69, 70, 95]. In their work, a goal-oriented method is proposed to derive workflow constraints. Although we use the policy-refinement tree structure to derive the low-level policies, which also include workflow constraints, each node of the policy refinement tree can be generally interpreted as a goal. Each node of the tree represents the intention, the objective, the plan that states the desired control of system's behavior. Nevertheless, our approach is capable of the automated policy refinement process.

LTL model checking-based refinement. Another related work is the work by Rubio-Loyola et al. [82]. Their work uses the goal refinement tree approach to manually refine the abstract policies. This refinement tree is *manually* constructed by security experts, who use the refinement pattern database. In contrast, our approach makes use of the formalized refinement patterns, which are acquired from the expert knowledge, to *automatically* construct the refinement tree.

Their work and ours use temporal logic model checking to perform the refinement of policy. In their work, they use model checking to find the execution traces in the model of the managed system, in which the concrete policies can be applied. On the other hand, our work uses model checking both for *applying* concrete policies in the particular execution traces as well as for *refining* policies. More precisely, we use model checking to perform *pattern matching* that proposes the refinement of an abstract policy.

Abduction refinement technique. The abduction-based refinement technique proposed by Bandara et al. [10] also uses a goal refinement tree approach to refine abstract policies. However, compared to our work, they use event calculus instead of temporal logic formalism. Furthermore, instead of using refinement patterns to refine the policies, they use the abduction reasoning technique. Their approach does not provide an automated refinement process and relies only on the knowledge of the policy administrator, who uses this approach. In contrast, our approach can perform the automated policy refinement process, which is supported by expert knowledge.

6.3 Pattern Matching

The pattern matching is performed in order to find a set of execution traces within the model, which match the description of the *context* and the *solution* of a pattern. Since the pattern's context is represented as a temporal logic formula and the workflow's behavior is represented as Kripke model, we can solve the pattern matching problem by employing the temporal logic model checking technique. Let ϕ_{ctx} be the temporal formula representing the context of the pattern. Furthermore, the Kripke model K should represent the workflow behavior. Let *states* be a function that maps each sequence into a set of its states. Thus, pattern matching should return a set of execution traces that is defined as follows:

$$\{\pi \mid (K, \pi \vDash \phi_{ctx}) \wedge (\forall z \in states(\pi) : prb \in L(z))\}.$$

However, a problem arises when we use the original temporal logic model checking to perform the pattern matching. This is due to the fact that the languages or terminologies used to specify the temporal formulas of the patterns' contexts and problems *differ* with the languages or terminologies used to label the states of the model.

To solve this problem, we already proposed an approach called description logic-based model checking, which is presented in Chapter 4. In this approach, the model and the formulas are translated into the description logic-based knowledge representation. The model is translated into a set of assertions stored in the ABox and the temporal logic formulas are translated into description logic concepts, which are stored in the TBox. In this case, the formulas ϕ_{ctx} are represented by the concept D_{ctx}. The atomic proposition prb is represented by the concept D_{prb}. The sequence of states π is translated into sequence of individuals σ. These translations are explained in Section 4.4 and Section A. Both ABox and TBox are the component of the knowledge base $\mathcal{KB}$. Thus, the description logic-based model checking process returns the set:

$$\Pi_{match} = \{\sigma \mid (\mathcal{KB} \vDash D_{ctx}(\sigma_0)) \wedge (\forall y \in states(\sigma) : \mathcal{KB} \vDash D_{prb}(y))\},$$

which contains the set of execution traces. Moreover, we use the algorithm introduced in Section 4.6 to retrieve the trace π.

6.4 Pattern Instantiation

If the set Π_{match} is not empty, then the solution of the pattern can be applied to these traces. Thus, depending on the solution, the abstract or the concrete solution should be instantiated. This process is also called *pattern instantiation*.

In the following we will discuss the instantiation of the abstract solution and then of the concrete solution.

6.4.1 Instantiating Abstract Solution

Instantiating an abstract solution is simple. The abstract solution is instantiated by adding new state labels, which are proposed by this solution, into each states of all execution traces, which match the context and the problem of the corresponding pattern.

Recall that the abstract solution is described as a set of atomic propositions $sln = \{prop_0, \ldots, prop_n\}$, where $prop_i$ is the atomic proposition representing the desired property. The instantiation of the abstract solution implies that each of the desired properties denoted by $prop_i$ shall hold along each execution trace $\pi \in \Pi_{match}$.

Thus, for each $prop_i$ of sln and for each $\pi \in \Pi_{match}$, all states of the model M appearing in trace π should be labeled with $prop_i$. These new labels represent the less abstract policies, which are classified either as protection plan or as protection event policies.

Pseudo code representing the instantiation of the abstract solution is presented as follows:

Input: abstract solution sln, trace π, labeling function of the Kripke model L
Output: labeling function L

```
1 foreach k ∈ sln do
2 |   foreach s ∈ states(π) do
3 |   |   L ← L ∪ {s ↦ {k}} ;
4 return L ;
```

6.4.2 Instantiating Concrete Solution

The instantiation process of a concrete solution differs from the instantiation of the abstract solution. In this step, there will be no addition of state labels

to the model. Instead, a concrete policy, such as the XACML policy, will be generated. Instantiating the concrete solution should also consider the context, which is also contained in the same pattern of the solution. We call this context as *concrete context*. Recall that the context describes a sequence of task executions, in which the problem exists. Thus, the syntax rule of the concrete context is defined as follows:

$$\phi_i ::= \begin{cases} T_i \wedge \mathbf{F}\ \phi_{i-1} & , \quad i > 0 \\ T_0 & , \quad i = 0 \end{cases},$$

where $i \in \mathbb{N}$. This formula describes the sequential execution of n tasks. $T_1 \ldots T_n$ are the task names ($T_i \in AP_{taskname}$). It should be noted that T_i preceeds T_{i-1} for all $i \in \mathbb{N}$. Such formula is satisfied by the execution trace:

$$s_{idx_0} \ldots s_{idx_1} \ldots s_{idx_n}$$

on which the following conditions hold:

$$0 \leq idx_0 < idx_1 < idx_2 < \ldots < idx_n,$$

$$\begin{aligned} T_n &\in L(s_{idx_0}), \\ T_{n-1} &\in L(s_{idx_1}), \\ &\vdots \\ T_0 &\in L(s_{idx_n}), \end{aligned}$$

and

$$T_n, \ldots, T_0 \in AP_{taskname}.$$

The atomic propositions $T_n, T_{n-1}, \ldots, T_0$ are the state labels of s_{idx_n},. . .,s_{idx_0}, respectively. Furthermore, these atomic propositions represent the task names. These atomic propositions are used to fill the parameter values of the XACML template. Thus, the XACML policy is generated by taking the task names as its parameter values.

6.4.3 An Example of Generating Separation of Duty Policy in XACML

As an example, we show the generation of separation of duty policy in XACML, which is adapted from the work by Crampton [27].

Let us consider an execution trace $\pi \in \Pi_{match}$. the trace π is a sequence of $s_1 s_2 s_3$. The states have the following labels:

$$\begin{aligned} s_1 &\mapsto \{createAppl, ApplySoD, preventFictiveLoanApplication\}, \\ s_2 &\mapsto \{checkAppl, ApplySoD, preventFictiveLoanApplication\}, \\ s_3 &\mapsto \{approveAppl, ApplySoD, preventFictiveLoanApplication\}, \end{aligned}$$

where $createAppl, checkAppl, approveAppl \in AP_{taskname}$ and $ApplySoD$, $preventFictiveLoanApplication \in AP_{securityproperty}$. As we can see, each state has a state label representing the task name and some state labels representing the security properties.

Suppose that we use the necessary common vocabulary of the tasks' names represented by an ontology, such that

$$\begin{aligned} createAppl &\equiv submit\_loan\_application \\ approveAppl &\equiv approve\_loan\_application. \end{aligned}$$

Thus, by using description logic model checking, we know that $\pi \models \phi_{ctx\_sod\_2}$, from

$$\phi_{ctx\_sod\_2} = E(submit\_loan\_application \wedge \mathbf{F}\ (approve\_loan\_application)).$$

To instantiate the concrete solution of the trace π, the refinement process generates the XACML policy by using the XACML policy template, which is stored as a concept. This was already discussed in Section 5.4.3.

The parameters of the template are filled by the atomic propositions, which are taken from the labels of the corresponding state s_1 and s_3. They are *createAppl* and *approveAppl*. It should be noted that only the labels, which represent the task names, are taken. Furthermore, each state has only a state label, which represents the task name. Figure 6.1 shows the XACML policy template, which has been already filled with the values *createAppl* and *approveAppl*.

6.5 Policy Refinement Algorithm

The policy refinement algorithm is an iteration of pattern matching and pattern instantiation steps, which are already introduced above. This iteration stops if there is no abstract policy that can be further refined by the patterns in the catalog *or* the leaves of the refinement tree represent the concrete policies. In the latter case, the refinement process is successfully finished. The policy refinement process is depicted in Algorithm 6.2.

```
<PolicySet ...  PolicySetId="Permission:Set" ...  >
  <Policy ...  PolicyId="permission:po:create" ...  >
    <Rule ...  >
    <Target>
      TaskID = "createAppl"
    </Target>
    <Obligations ...  >
      <Obligation FulfillOn="Permit" ...  >
        addDenyPermissionToSubjectBlacklist
        ("approveAppl")
      </Obligation>
    </Obligations>
  </Policy>
  .
  .
  .
</PolicySet>
```

Figure 6.1: The generated XACML policy

Input: pattern database Π,Kripke Model M
Output: XACML Policies XPL,Refinement Tree T

$T \leftarrow \emptyset$; **while** *true* **do**
  $Match \leftarrow \emptyset$;
  foreach $pat \in \Pi$ **do**
    $Tr \leftarrow getTraces(\texttt{parseTreeOf}(pat.ctx \wedge pat.prb), M)$;
    foreach $t \in Tr$ **do**
      $ProposedSolution \leftarrow \emptyset$;
      foreach $s \in sln(pat)$ **do**
        if $NodeIsNotFound(T, s, t)$ **then**
          $ProposedSolution \leftarrow ProposedSolution \cup \{s\}$;
      $Match \leftarrow Match \cup \{(t, ProposedSolution)\}$
  if $Match = \emptyset$ **then**
    return;
  foreach $n \in Match$ **do**
    foreach $l \in n.ProposedSolution$ **do**
      if *l.s is abstract solution* **then**
        $addLabel(l.s, n.t)$;
        $T.addNodeToLeaf(l.s, n.t)$;
      else if *n.s is auth. constraint solution* **then**
        $generateXACMLAuth(l.s, n.t)$;
      else if *p is SoD solution* **then**
        $generateXACMLSoD(l.s, n.t)$;

Figure 6.2: The algorithm for pattern refinement

Pattern Matching. `(line 3-10)`

Since we have the formalization of the workflow, its policies and the policy refinement patterns, we could perform an automated pattern matching by using model checking techniques. This is done by evaluating the LTL formulas of context and problem of every pattern in the catalog over the model that represents both the workflow and its policies.

Basically, the pattern matching, as already explained in Section 6.3, is performed by $getTraces(p, M)$, which takes a pattern and the model as its input and returns some the execution paths. It returns the execution traces, which match the problem and the context of pattern p. The loop between lines 7-9, which calls the $NodeIsNotFound(T, s, t)$ function, checks whether the matched patterns are already instantiated or not.

Pattern Instantiation. `(line 14-22)`

We differentiate between the instantiation of the abstract solutions and of the concrete solutions. The instantiation of the abstract solutions only involves the creation of new state labels within the execution path, which satisfies the restricted CTL* formula of context and problem. On the other hand, instantiating the concrete solution requires the generation of the workflow constraints.

XACML Policy Generation. `(line 19-22)`

The constraint generation step is considered as a special case of the pattern instantiation step that has a concrete solution. This process takes the execution path and the pattern name as the input, and generates an XACML policy as its output. In our case, we consider in generating XACML policy that specifies either the authentication constraints or the authorization constraints. An authentication constraint restrict the execution of certain tasks only to subjects, which have a specific assertion about their authentication method. Such constraint can be expressed in XACML. On the other hand, we consider only the authorization constraints, which specify the separation of duties.

The generated XACML policy consists of a target and a rule. The target of this policy describes the situation, in which the policy applies. The target specifies the name of the task (resource) and the requesting subject, which has an attribute assertion about its authentication method. The rule part of this policy states the permission of the specified subject to execute this task.

The input of this step, which consists of an execution path and a pattern, determines the target of the generated XACML policy. The execution path satisfying the context has only one task, the name of which is inserted into

the target of the policy. The concrete solution of the pattern is also inserted into the subject's attribute assertion of the target.

6.6 Example

In this section we will present a trivial example of the policy refinement process.

6.6.1 Workflow and its High-Level Policies

First, we will present the set of states, state transitions and state labels representing the workflow behavior as follows:

$$\begin{aligned} W &= \{s_1, s_2, s_3, s_4\} \\ I &= \{s_1\} \\ RL &= \{(s_1, s_2), (s_2, s_3), (s_3, s_4)\} \\ L_{wf} &= \{s_1 \mapsto \{SubmitLoanAppl\}, s_2 \mapsto \{CheckLoanAppl\}, \\ &\quad s_3 \mapsto \{ApproveLoanAppl\}, s_4 \mapsto \{SendPayment\}\}. \end{aligned}$$

Additionally, the stakeholders specify their high-level policies by labeling *all* states within the model with the atomic propositions representing their desired properties. In this example, the stakeholders intend to achieve a secure loan application process. This property is denoted by the atomic proposition
$secureLoanApplProcess$. Thus, the following set of mappings represents the high-level policies:

$$\begin{aligned} L_{pol} &= \{s_1 \mapsto \{SecureLoanAppProc\}, s_2 \mapsto \{SecureLoanAppProc\}, \\ &\quad s_3 \mapsto \{SecureLoanAppProc\}, s_4 \mapsto \{SecureLoanAppProc\}\}. \end{aligned}$$

The set of atomic propositions is defined as:

$$\begin{aligned} AP &= \{SubmitLoanAppl, CheckLoanAppl, ApproveLoanAppl, \\ &\quad SendPayment, SecureLoanAppProc\}. \end{aligned}$$

Finally, we define the Kripke model representing the workflow's behavior and its high-level policies as:

$$K = \langle W, RL, L_{wf} \cup L_{pol}, AP \rangle$$

Figure 6.4 graphically depicts the Kripke model of the workflow in the initial phase. As we can see, the state label $SecureLoanAppProc$ represents

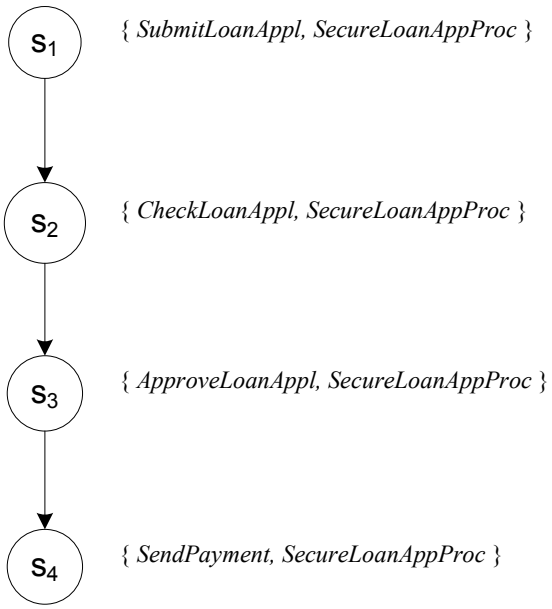

Figure 6.3: Kripke model of the workflow in the initial phase

a high-level policy of the stakeholders. However, we use the policy refinement tree to structurally represent the policies. Let T_{pol} be the policy refinement tree. Initially, T_{pol} has the following components:

$$\begin{aligned}
\omega & \\
N_{spec} &= \{(\{s_1, s_2, s_3, s_4\}, secureLoanAppProc)\} \\
N_{abs} &= \emptyset \\
N_{con} &= \emptyset \\
AndGate &= \{(\omega, \{(\{s_1, s_2, s_3, s_4\}, secureLoanAppProc)\} \\
OrGate &= \emptyset \\
app &= \emptyset
\end{aligned}$$

The policy refinement tree is graphically depicted in Figure 6.4.

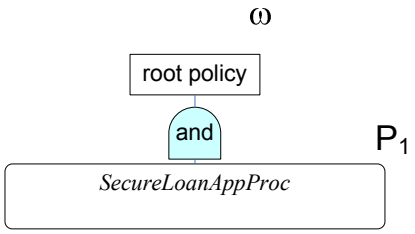

Figure 6.4: Refinement tree in the initial phase

6.6.2 Refinement Patterns

As for the example, we present three refinement patterns, two abstract refinement patterns and a concrete refinement pattern. For the sake of simplicity, we will not present them in form of description logic notation, instead, as tuples.

The first pattern is referred to as `Pat1` or to as "Secure Loan Application Process" (slap) pattern. It has the purpose to refine "Secure Loan Application Process" policy into two less-abstract policies, namely "Prevent Fictive Loan" and "Prevent Unauthorized Access in Loan Application". This pattern is denoted by the tuple $(\phi_{slap\_ctx}, slap\_prb, slap\_sln)$. It should be noted that the abbreviation "LAP" in the atomic propositions stands for "loan application process". They are defined as follows:

$$\begin{aligned} \phi_{slap\_ctx} &= E(SubmitLoanAppl \wedge \mathbf{F}\ (CheckLoanAppl \wedge \\ &\quad \mathbf{F}\ (ApproveLoanAppl \wedge \mathbf{F}\ SendPayment))) \\ slap\_prb &= SecureLoanAppProc \\ slap\_sln &= \{preventFictiveLoan, preventUnauthAccessInLAP\} \end{aligned}$$

The second pattern is referred to as `Pat2` or to as "Separating Check Loan and Approve Loan Tasks to Prevent Fictive Loan" (stpf). It has the purpose to derive the necessary protection event, which can prevent fraud in the loan application process. In this case, this pattern suggests the separation of the two tasks, namely "check loan" and "approve loan". This pattern is denoted by the tuple $(\phi_{stpf\_ctx}, stpf\_prb, stpf\_sln)$. They are defined as follows:

$$\begin{aligned} \phi_{stpf\_ctx} &= E(SubmitLoanApp \wedge \mathbf{F}\ (ApproveLoanAppl)) \\ stpf\_prb &= preventFictiveLoan \\ stpf\_sln &= \{applySoD\} \end{aligned}$$

The third pattern is a concrete pattern. It is referred to as `Pat3` or to as "Generating XACML Policy for Separation of Duty" (asod). This pattern is denoted by the tuple $(\phi_{asod\_ctx}, asod\_prb, \mathsf{XacmlSoD})$. The context and the problem are defined as follows:

$$\begin{aligned} \phi_{asod\_ctx} &= E(SubmitLoanApp \wedge \mathbf{F}\ (ApproveLoanAppl)), \\ asod\_prb &= applySoD, \end{aligned}$$

where the solution of the pattern is a template of an XACML policy. The template is already presented in Figure 5.1 and Example 20.

6.6.3 Refinement Steps

Pattern Matching - 1st Iteration. In the pattern matching step of the first iteration, the context and the problem of `Pat1` should match the following set of traces:

$$Tr_{match\_1} = \{s_1 s_2 s_3 s_4\},$$

due to the fact that $\forall tr \in Tr_{match} : (tr \vDash \phi_{slap\_ctx}) \wedge \forall z \in states(tr) : slap\_prb \in L(z)$.

Pattern Instantiation - 1st Iteration. Based on the set Tr_{match_1}, the solution of pattern `Pat1` should be instantiated. In this step, all states of each trace in Trans_{match_1} should be given the label proposed by $slap\_sln$. In this case, the new labels are $preventFraudInLoanAppl$ and$preventUnauthAccessInLoanAppl$. Thus, the labeling set L of the model K will be added with new state labels:

$$\begin{aligned} L' = L \cup \{ & s_1 \mapsto \{preventFictiveLoan, preventUnauthAccessInLAP\}, \\ & s_2 \mapsto \{preventFictiveLoan, preventUnauthAccessInLAP\}, \\ & s_3 \mapsto \{preventFictiveLoan, preventUnauthAccessInLAP\}, \\ & s_4 \mapsto \{preventFictiveLoan, preventUnauthAccessInLAP\}\} \end{aligned}$$

This model is depicted in Figure 6.5.

Moreover, some new policies and gates are added into refinement tree T_{pol}.

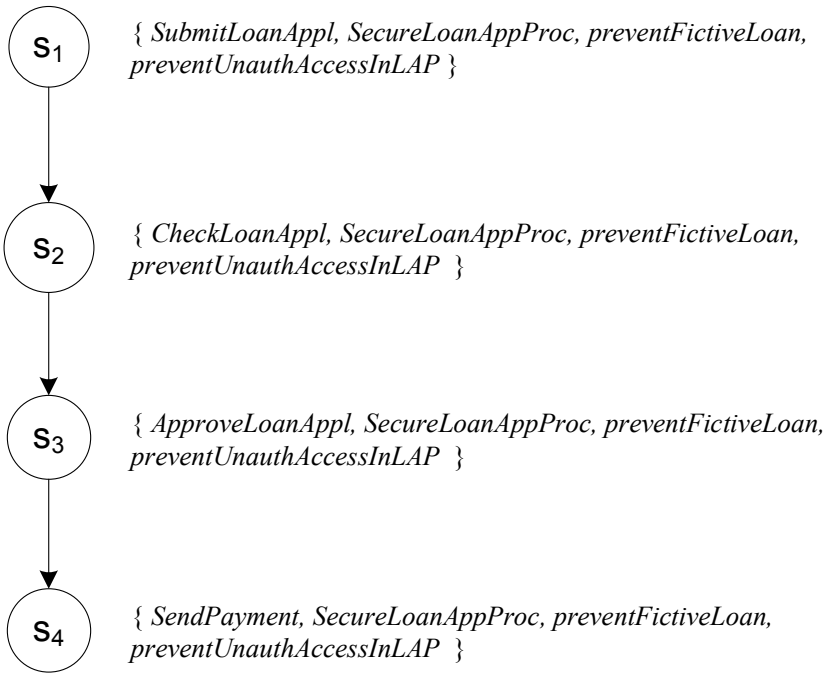

Figure 6.5: Kripke model of the workflow after the first iteration

$$
\begin{aligned}
N'_{abs} &= N_{abs} \cup \{(s_1s_2s_3s_4, SolutionOfPattern1),\\
&\quad (s_1s_2s_3s_4, preventFictiveLoan),\\
&\quad (s_1s_2s_3s_4, preventUnauthAccessInLAP)\}\\
N'_{con} &= \emptyset\\
AndGate' &= AndGate \cup \{\\
&\quad ((s_1s_2s_3s_4, SolutionOfPattern1),\\
&\quad \{(s_1s_2s_3s_4, preventFictiveLoan)\}),\\
&\quad ((s_1s_2s_3s_4, SolutionOfPattern1),\\
&\quad \{(s_1s_2s_3s_4, preventUnauthAccessInLAP)\})\}\\
OrGate' &= OrGate \cup \{((\{s_1, s_2, s_3, s_4\}, secureLoanAppProc)\\
&\quad , \{(s_1s_2s_3s_4, SolutionOfPattern1)\})\}
\end{aligned}
$$

As we can see, the policy P'_1 acts as the intermediary policy between the abstract policy and the sub policies, which are proposed by the pattern `Pat1`. It should be noted that this kind of policy is connected with the

abstract policy and the generated sub policies by the `OR` gate and `AND` gate, respectively. It means that the solution is an *alternative* to refine the abstract policy, and fulfilled only by enforcing *all* of the sub policies. The current refinement tree is shown in Figure 6.6.

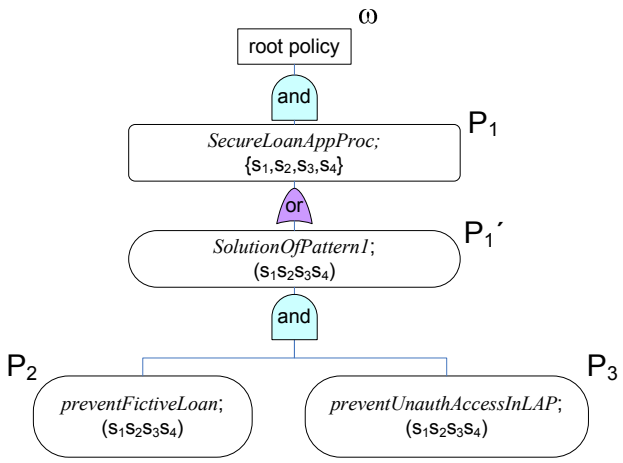

Figure 6.6: Refinement tree after the first iteration

Pattern Matching - 2nd Iteration. Pattern `Pat1` will be ignored in the second iteration of the pattern matchin process, due to the fact that it already has a match with the same execution path found in the first iteration. In this iteration, the context and the problem of `Pat2` will match the following set of traces:

$$Tr_{match\_2} = \{s_1 s_2 s_3\},$$

due to the fact that $\forall tr \in Tr_{match} : (tr \vDash \phi_{stpf\_ctx}) \wedge \forall z \in states(tr) : stpf\_prb \in L(z)$.

Pattern Instantiation - 2nd Iteration. Based on the set Tr_{match_2}, the solution of pattern `Pat2` should be instantiated. In this step, all states of each trace in Tr_{match_2} should be given the label proposed by ϕ_{stpf_sln}. In this case, the new label is $applySoD$. Thus, the labeling set L of the model K will be added with new state labels:

$$\begin{aligned} L'' \;=\; & L' \cup \{s_1 \mapsto \{applySoD\}, \\ & s_2 \mapsto \{applySoD\}, \\ & s_3 \mapsto \{applySoD\}\} \end{aligned}$$

This model is depicted in Figure 6.7.

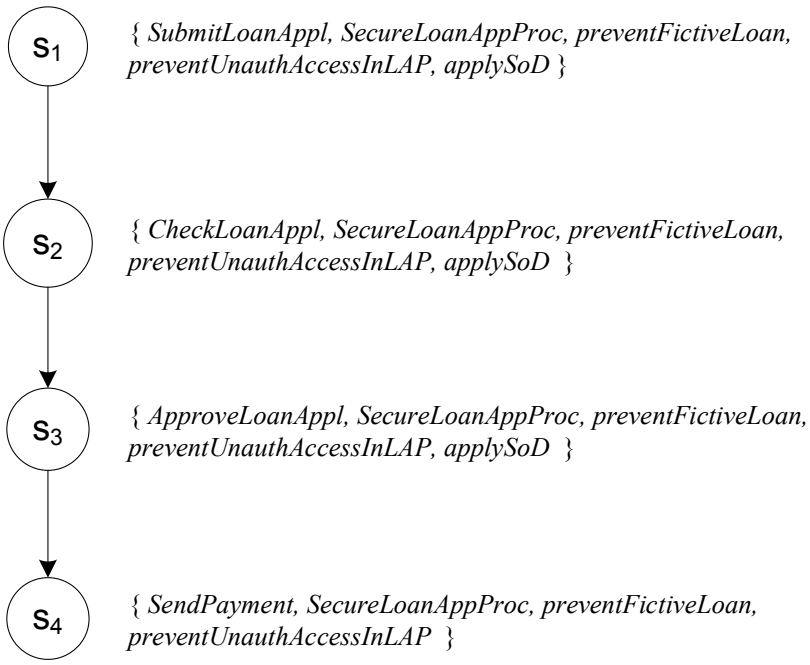

Figure 6.7: Kripke model of the workflow after the first iteration

Moreover, the refinement tree T_{pol} will be added with new nodes and relationships:

$$\begin{aligned}
N''_{abs} &= N'_{abs} \cup \{(s_1s_2s_3s_4, SolutionOfPattern2), \\
&\quad (s_1s_2s_3, apply_separation_of_duty)\} \\
N''_{con} &= \emptyset \\
AndGate'' &= AndGate' \cup \{((s_1s_2s_3s_4, SolutionOfPattern2), \\
&\quad \{(s_1s_2s_3, apply_separation_of_duty)\})\} \\
OrGate'' &= OrGate' \cup \{((s_1s_2s_3s_4, preventFictiveLoan), \\
&\quad \{(s_1s_2s_3s_4, SolutionOfPattern2)\})\}
\end{aligned}$$

Again, we see that the policy P'_2 acts as the intermediary policy between the abstract policy and the sub policies, which are proposed by the pattern `Pat2`. The current refinement tree is shown in Figure 6.8.

Pattern Matching - 3rd Iteration. Again, the patterns `Pat1` and `Pat2` will be ignored. In the third iteration, the context and the problem of `Pat3`

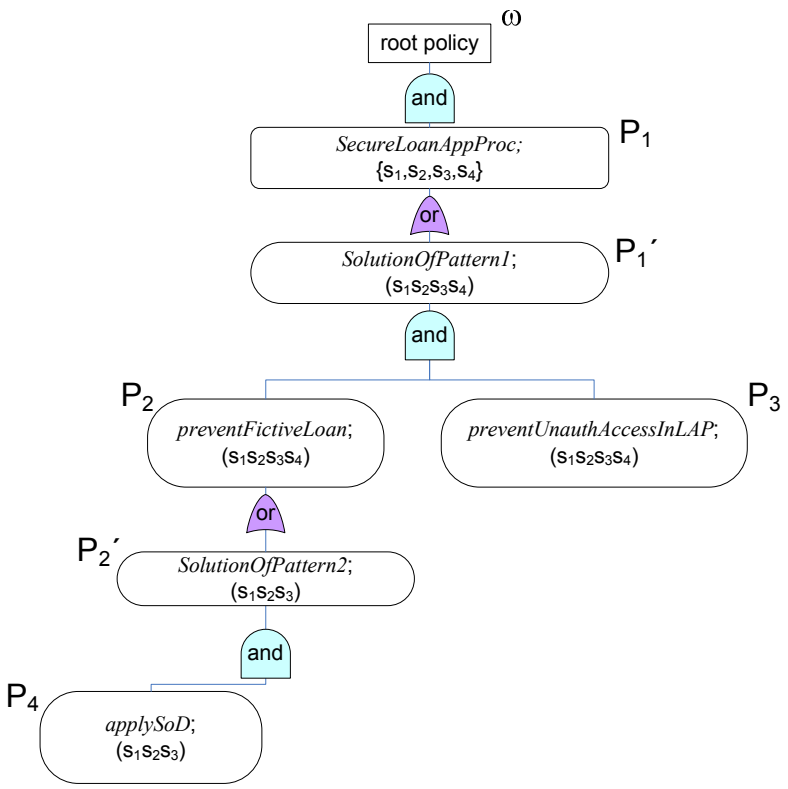

Figure 6.8: Refinement tree after the second iteration

will match the following set of traces:

$$Tr_{match\_3} = \{s_1 s_2 s_3\},$$

due to the fact that $\forall tr \in Tr_{match} : (tr \vDash \phi_{asod\_ctx}) \wedge \forall z \in states(tr) : asod\_prb \in L(z)$.

Pattern Instantiation - 3rd Iteration. Based on the set Tr_{match_3}, the solution of pattern `Pat3` should be instantiated. In this step, an XACML policy will be generated and the refinement tree T_{pol} will be added with new nodes and relationships:

$$\begin{aligned}
N'''_{abs} &= N''_{abs} \cup \{(s_1 s_2 s_3, SolutionOfPattern3)\} \\
N'''_{con} &= N''_{con} \cup \{(s_1 s_2 s_3, XacmlSoD)\} \\
AndGate''' &= AndGate'' \cup \\
&\quad \{((s_1 s_2 s_3, SolutionOfPattern3), \{(s_1 s_2 s_3, XacmlSoD)\})\} \\
OrGate''' &= OrGate'' \cup \{((s_1 s_2 s_3, apply_separation_of_duty), \\
&\quad \{(s_1 s_2 s_3, SolutionOfPattern3)\})\}
\end{aligned}$$

Thus, there is no change made in the Kripke model. The current refinement tree is shown in Figure 6.9.

Since the generation of the separation of duty policy in XACML is already discussed in Section 6.4.3, we will not discuss further about the XACML policy generation.

6.7 Discussion

In this section, we give our analysis and limitations of the policy refinement process. Section 6.7.1 and 6.7.2 present some scenarios, which could occur in the policy refinement process. The rest of this section presents the other limitation of this research work.

6.7.1 Incomplete Policy Refinement Due to the Lack of Knowledge

Let us assume that the refinement patterns database does not have the sufficient patterns. This can cause that the refinement process cannot derive the enforceable policies. In this case, the refinement process will prematurely terminate and the policy refinement tree would not be completely built. One

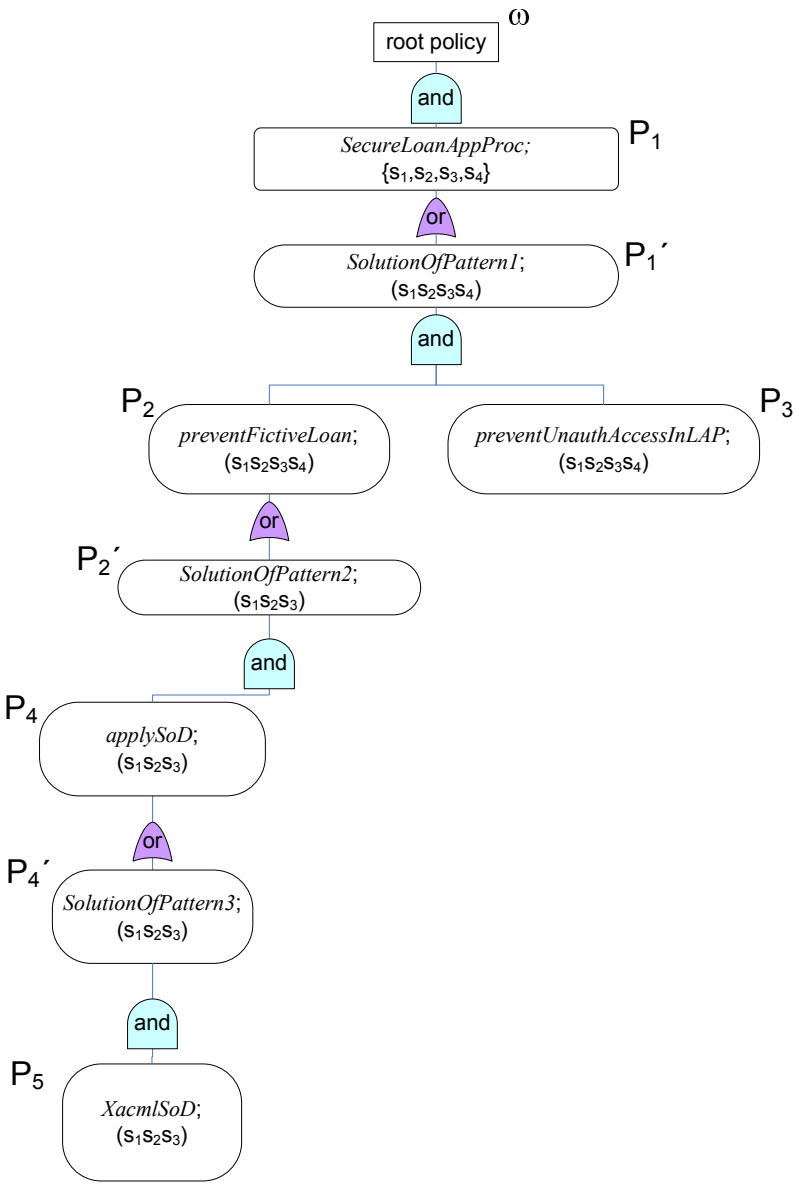

Figure 6.9: Refinement tree after the third iteration

can see that the leaves of the policy refinement tree are not from the type of the concrete policies.

In this case, the tool can only partially generate workflow policies. Moreover, the generated workflow policies cannot fulfill the high-level policies. To solve this problem, we need to acquire more refinement patterns.

6.7.2 Refinement with Bad Pattern Quality

Let us assume that the database has some refinement patterns with bad quality. The refinement process finishes without any error and the policy refinement tree is completely built.

Although the policies are completely generated, these generated policies cannot fulfill the high-level policies. This problem is known if the generated policies are verified or the administrator discovers any malicious behavior during monitoring.

6.7.3 Limitations of the Generated Policies

XACML Policies Limitations. In our case, we only consider in deriving concrete policies, which specify authorization constraints and/or authentication constraints. It is well known that XACML policies cannot be used to specify policies, which consider the status nor the history of the access control. For example, XACML policies cannot be used to specify chinese wall policies.

Lack of Expressiveness in Specifying Semantic and Security Constraints. Biskup and Sprick [16] proposed an approach, which allows us to specify semantic and security constraints. These constraints could be specified for a workflow. Our approach, however, does not consider such constraints. We specify our workflow constraints either as a set of state labels or as enforceable policies. On the other hand, their approach specify the constraints as logical formulas.

6.8 Summary

In this chapter we learnt how building blocks are used together to realize the description logic-based model checking. We presented the algorithm to build the policy refinement tree by using the refinement patterns stored in the database.

The pattern matching is solved by employing the model checking technique, which is modified in order to fulfill our need. The pattern instantiation process is either the addition of new state labels in the Kripke model or the generation of the XACML policies.

In the next chapter, we will present the architecture of the policy refinement tool, which is developed as a proof of concept of this work.

Chapter 7

The Policy Refinement Tool

7.1 Introduction

The previous chapters presented the concepts required in realizing the automated and pattern-based workflow policy refinement process. Moreover, they also discuss the theoretical foundation needed for our concepts. In this chapter, we will discuss the implementation of the automated policy refinement tool. In implementing the refinement tool, we strive to maximally utilize existing standards and tools.

Outline. The chapter is divided into three parts. The first part discusses the standards and the tools that are used to realize the policy refinement tool. The second part explains the architecture of the refinement tool. The last part of this chapter discusses the contributions of our work in Sicari project.

7.2 Standards and Tools

Our goal in implementing the refinement tool is to maximally utilize existing standards, libraries or tools. With this goal, we can facilitate the extensibility of the refinement tool and also reduce the implementation effort. In the following, we will present the standards and tools that are used in the implementation of the refinement tool.

7.2.1 Web Ontology Language (OWL)

Web Ontology Language (OWL) is the standard for representing
the machine-interpretable knowledge in form of a mark-up language stored

and transferred over the internet, which is also known as the web.

For the past decades, several researchers are aware that the web would benefit from some structure and explicit semantics for some of its content. This clearly enables the interpretation of web content by machines and obviously facilitates the conceptual search of web content. In comparison to the traditional textual or syntactical search, the conceptual search allows the retrieval of a set of web instances having the same interpretation, even if these instances are represented in different texts.

The development of the OWL standard is based on two existing standards, namely DAML (DARPA Agent Mark-up Language) and OIL (Ontology Inference Layer). The DAML standard was established from the DARPA Program in 1999 [8]. It aims to utilize agents and programs to interpret, search and process web content [49]. The language of the DAML standard is written in the form of a mark-up language. At the same time, the OIL standard was being developed. It has also the same goal with the DARPA standard. However, the developers of OIL placed a stronger emphasis on formal foundations. Thus, the semantics of the OIL language can be specified via a mapping to the semantics of the Description Logic $\mathcal{SHOQ}(\mathcal{D})$ [41, 51].

Consequently, the two approaches were combined and resulted in the DAML + OIL. The DAML+OIL is represented by using the RDFS standard. Later on, the DAML + OIL was renamed into OWL, and currently there are three classes of OWL languages: (i) OWL Lite, (ii) OWL DL, and (iii) OWL Full. In our work, we use the OWL DL language, but has more expressive power ($\mathcal{SHOQ}(\mathcal{D})$) than OWL Lite, but it has a more efficient reasoning algorithm compared to OWL Full. To specify, edit and manipulate the OWL language, we use the Protege Ontology Editor.

7.2.2 Protege Ontology Editor

The Protege Ontology Editor [66] is a license-free and open-source software that can be freely downloaded[1]. The purpose of Protege is to create, manage, view and edit the knowledge represented in OWL language by providing a graphical user interface. It also has the DIG interface, which allows the user to attach existing description logic reasoners with the Protege Ontology Editor. This has the advantage, that one can perform the query directly from the graphical user interface of the Protege Ontology Editor.

We mainly use the Protege Ontology Editor to specify the refinement patterns and to perform experiments related to description logic-based model checking.

[1] `http://protege.stanford.edu/`

7.2.3 Extensible Access Control Mark-up Language

The Extensible Access Control Mark-up Language is a standard developed by OASIS-OPEN consortium [71]. The goal of this standard is to provide an open standard for specifying access control policies, which is free of license and can be freely extended by the users. The access control policies are intended to control the permission to access web resources, which are specified by a Uniform Resource Identifier (URI). The current version of the XACML standard is Version 2.0. Although the next revision, Version 3.0, is currently being developed.

7.3 The Modules of the Policy Refinement Tool

In this section we will explain the architecture of the refinement tool and its modules. The whole architecture can be seen in Figure 7.1.

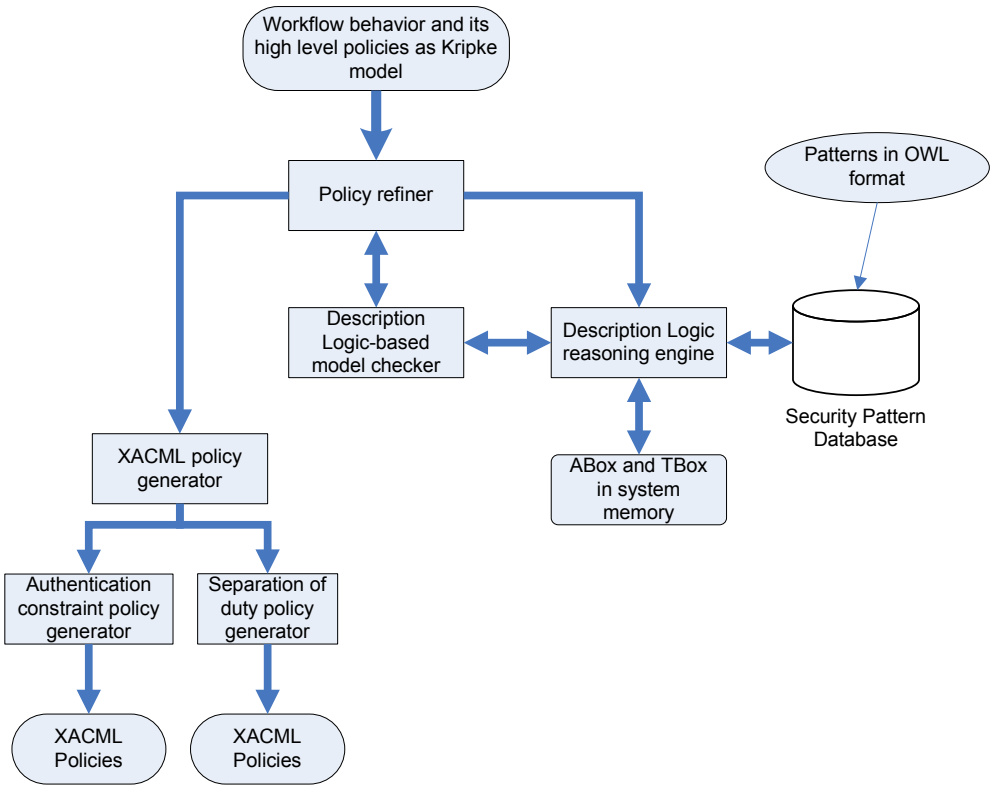

Figure 7.1: The architecture of the automated policy refiner

7.3.1 Policy Refiner

This is the main module of the policy refinement tool. It takes the Kripke model of the behavior of workflow and high-level policies as the input and generates the policy refinement tree and concrete policies as the output. The Kripke model is forwarded to the description logic reasoning engine, which stores the Kripke model in the Abox. In this prototype, the input of this module is written in OWL format.

The objective of this module is to refine the high-level workflow policies by using refinement patterns. This module performs the iteration of the refinement step, which consists of the pattern matching and pattern instantiation processes. It consults the description logic-based model checking to find the refinement patterns, which can refine the – currently unrefined – abstract workflow policies (see Section 6.3). This is done by performing model checking for each pattern in the database. Obviously, this process is the most time consuming part of the policy refinement process. This is due to tha fact that the model checking process is performed for all refinement patterns in the database, which are repeated for each refinement step. If patterns are found, the solutions of the patterns are instantiated by adding the new state labels in the Kripke model stored in the Abox (see Section 6.4). This refinement step is repeated until all abstract policies are refined.

Additionally, this module also builds the policy refinement tree. Representing the refinement tree is relatively straightforward. It is represented by the graph data structure, which is stored internally by the policy refiner module.

The Graphical User Interface of the Policy Refiner can be seen in Figure 7.2.

On the left pane, we see the policy refinement tree. This hierarchical structure is represented by using the explorer tree. The right pane presents the policy. It shows the execution path and the desired security property, which should hold in the execution path.

7.3.2 Description Logic reasoning engine

We conduct our experiment with several existing description logic reasoning engines, such as Pellet reasoner [74] and Racer [47] to realize this module. We use the Protege-OWL API [94] to interact with the reasoner.

The task of this module is to manage the knowledge base represented in the TBox and the Abox, which are stored in the system's memory. Furthermore, it also accepts the assertions and queries sent by the policy refiner and description logic-based model checker modules, respectively. The pol-

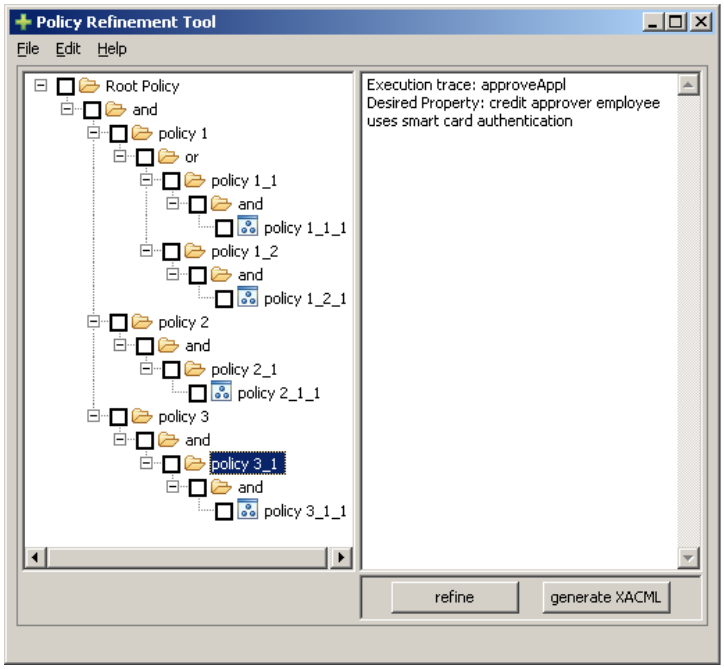

Figure 7.2: The user interface of policy refiner tool

icy refiner sends the assertions in order to represent the Kripke model in the Abox. The description logic-based model checker sends the instance checking queries in order to perform the model checking process.

In the initialization phase, the module reads the refinement patterns database stored in the OWL file and loads them into the TBox stored in the system's memory.

7.3.3 Description Logic-based model checker

Although this module can perform the model checking process, the main task of the module is to perform the automated pattern matching on behalf of the policy refiner module. For each refinement step, the module receives the query from the policy refinement module. The query asks for a set of the tuples containing the pattern's identifier and a set of execution traces, in which the pattern's solution can be applied. To answer this query, the module performs the pattern matching as described in Section 6.3 by means of description logic-based model checking.

7.3.4 XACML Policy Generator

This module facilitates the generation of the enforceable XACML policies according to the policy refinement tree, which is generated by the policy refiner module. Clearly, the leaves of the policy refinement tree denote the enforceable low-level policies. However, we need to generate the corresponding machine interpretable and enforceable policies, such as XACML policies, in order to achieve the desired high-level policies. The module takes the refinement tree as its input.

The module, however, does not generate the XACML policies itself, but it acts as a proxy between the policy refiner and the real XACML policy generators. Thus, the module implements the abstract factory pattern [42]. It determines, which kinds of concrete solutions are being instantiated and forwards them to the corresponding XACML generator modules.

In our work, we consider only two kinds of policies that should be generated. They are authorization constraint policy and separation of duty policy. According to Figure 7.1, we have two modules called *authentication constraint policy generator* and *separation of duty policy generator* that interact with the XACML policy generator module. This is due to the fact that we consider only two mechanism refinement patterns that generate the authentication constraint policy and the separation of duty policy, respectively. Obviously, adding some new mechanism refinement patterns should also integrate new constraint generator modules in this architecture.

To generate the XACML policy, we use the Xerces library provided by the Apache XML Project [99] and use the XML schema file defined in the XACML specification [71].

7.4 Contributions to Sicari Project

The result of our research work is placed within the sub project policies, which deals with the realization of the policy-based system management on the Sicari platform. The architecture of the policy-based system management is presented in Figure 7.3. Policy Enforcement Point, Policy Decision Point, Policy Validation and Policy Refinement Tool modules are developed within the sub project “Policies”. These modules are depicted in Figure 7.3. It should be noted that we use XACML standard to specify our enforceable policies.

The Policy Enforcement Point monitors each access requests on sensitive resources and consults with the Policy Decision Point, which determines the permission or denial of the access requests.

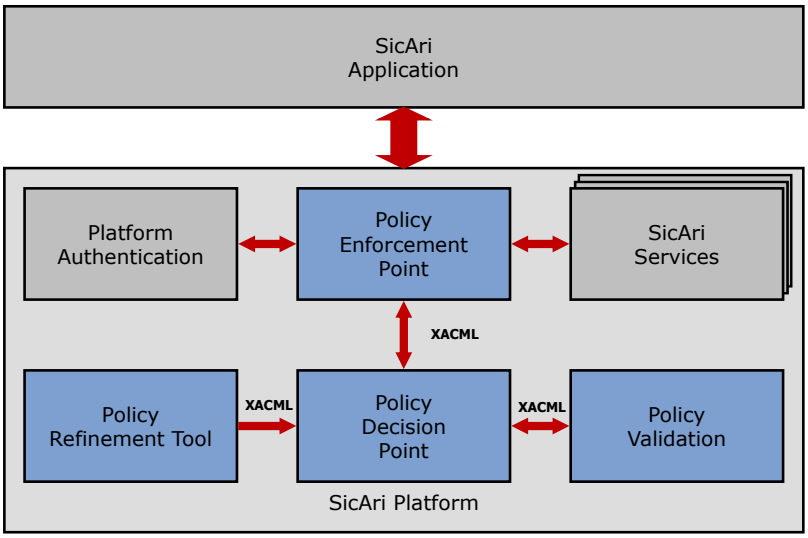

Figure 7.3: The architecture of the Sicari platform

The Policy Validation Point checks whether the generated enforceable policies comply with the stakeholders' high-level policies. This module makes use of the Simple Homomorphism Verification Tool (SHVT), which is developed by Fraunhofer SIT and provides a formal tool to verify the properties of the modeled system. By using the SHVT tool, we aim for verifying the compliance between the generated workflow security policies and the high-level policies. To achieve this goal, one should construct the accessibility graph. This graph represents the behavior of the enforcement system, which accepts the generated workflow policies. The query is specified according to the high-level workflow policies. If the query finds the desired states, then the high-level policies comply with the behavior of the enforcement system, which accepts the generated workflow policies. It should be noted that both of the tasks, the construction of the accessibility graph and the specification of the query are specified manually.

The last module, which is the proof of concept of the research presented in this thesis, aims at automatically generating the enforceable policies from the stakeholders' high-level policies by using expert knowledge stored in machine-interpretable representation.

7.5 Summary

In the first half of this chapter, we studied the standards and tools that are used to realize the refinement tool. We strived to use the existing standards and tools in order to minimize the implementation effort and also to enable the modularity and extensibility of the refinement tool.

In the second half of this chapter we present the architecture and its modules. The architecture and its modules should give the overview to the reader, who may want to extend or modify the existing implementation.

Chapter 8

Summary and Outlook

The importance of policy-based system management increases as the complexity of the managed system arises. The policy-based system management approach significantly helps the stakeholders to control the behavior of the managed system, which should follow the desired behavior of the stakeholders. In this case, the system should follow the protection intent of the stakeholders, which is specified in terms of high-level policies. Since these policies cannot be directly enforced, the security experts should manually refine the high-level policies into low-level, enforceable policies, which can be directly enforced by the machine.

However, the manual refinement process posses some disadvantages. One of the advantages is that the manual refinement process is susceptible to human errors. A simple error in the derivation of sub policies in the early stages of the refinement process could lead into critical flaws that appear in the low-level, enforceable policies. The other disadvantage of the manual refinement process is that it requires security expert knowledge in refining policies. As the matter of fact, security experts are rare to be found.

In our study, we take the workflow scenario as our proof of concept. Obviously, one may also apply this approach to automate the policy refinement process in any other policy-based system management (firewall routers, web servers, etc) provided that the behavior of the considered systems can be formally represented as Kripke models and the corresponding refinement patterns exist.

We will summarize the results of this work, which are structured in the following chapters:

Chapter 2: Workflows and Policies. Since we make use of workflow scenario as a proof of concept, we described in this chapter the formal representation of the workflow by using a Kripke model. We then presented the

method to specify workflow policies as the state labels of the corresponding Kripke model. In this chapter, we also presented the classification of the workflow policies according to their abstraction levels. They are, namely, specified workflow policies, abstract workflow policies and concrete workflow policies. It should be noted that the policies in different abstraction levels have dependencies. In other words, the fulfillment of a policy depends on the fulfillments of its decomposed sub policies. This motivated us to construct the policy refinement tree, which was presented in the next chapter.

Chapter 3: Policy Refinement Tree. In the previous chapter we studied the different abstraction levels of workflow policies. These policies were represented as the state labels of the Kripke model. However, this kind of representation aggravates the analysis of the fulfillment of the high-level policies. Therefore, this chapter presented the definition of hierarchical structure of the workflow policies as a policy refinement tree. Every policy refinement tree corresponded to a Kripke model. The structure of the policy refinement tree was adopted from the fault tree analysis method. In this chapter, we also presented the formal representation of the policy refinement tree.

Chapter 4: Description Logic-based Model Checking. The first building block of the automated policy refinement process was concerned with the novel approach in combining the temporal logic formalism with the description logic formalism. We used the temporal model checking approach to perform the automated pattern matching step. Since we had the Kripke model representing the workflow's behavior and its policies, we were able to perform the pattern matching step if we represent the patterns in terms of temporal formulas. However, due to the fact that the atomic propositions used to construct the Kripke model and to specify the temporal formulas were different, traditional model checking approaches could not be used to perform the automated pattern matching step. We aimed to enable the pattern matching step, which used the model checking approach, by combining temporal logic formalism with the description logic formalism. Thus, we called this approach description logic-based model checking.

Chapter 5: Refinement Patterns. This chapter presented the second building block of the automated policy refinement process. In this chapter we presented the definition of the refinement patterns adopted from the work by Alexander. In this chapter we also presented the categorization of the refinement patterns and the formalization of the refinement patterns.

Chapter 6: Policy Refinement Process. This chapter defined the algorithm that realized the automated policy refinement process. This algorithm used of the previously defined building blocks, namely the description logic-based model checking and the refinement patterns.

Chapter 7: Architecture. This chapter presented the tools and standards that were used to implement the automated policy refinement tool.

Scientific Contribution In this work, we investigated the approach to automate the policy refinement process, which is supported by expert knowledge. In order to realize our main goal, we formalized the behavior of the considered system and its high level policies and also presented the formal representation of the refinement patterns, which capture the expert knowledge.

The expert knowledge-based policy refinement process performs several iterations, which consist of *pattern matching* and *pattern instantiation*. The most challenging part is the pattern matching problem, since it searches for a set of refinement patterns, which can refine the abstract policies.

To enable the automated pattern matching we extended the traditional *model checking* technique by combining the temporal logic formalism with description logic formalism. This approach enabled *ontology-supported temporal reasoning* without introducing additional computation complexity to the existing description logic reasoning engine. It also allows us to perform *both semantic and temporal* reasoning on a semantic web reasoning engine.

8.1 Future Work

In the following we present the open issues that can be used as future work.

- *Refinement pattern engineering*
  We are aware that the quality of the refinement patterns depends on the pattern mining process. To improve the quality of the refinement patterns, appropriate methods and concepts should be developed.

- *Choosing the best set of generated enforceable policies*
  Our concept of the automated policy refinement process generates all possible set of enforceable policies. However, only a subset of this generated policies are needed to be enforced in order to fulfill the specified abstract policies.

In the field of fault tree analysis, there is an approach, which solves the similar problem. The aim of this approach is to find a subset of basic events (leaves of the fault tree) that can cause the main failure to occur. This approach is called *minimal cut set algorithm*.

In our case, however, we should consider several factors in order to find a subset of enforceable policies, which satisfy the specified policies. These factors are: (i) the cost of enforcing enforceable policies and (ii) the cost of loss due to the unfulfilled policies.

To give an overview of this problem, let us consider the matrix in Figure 8.1. This matrix depicts the policy enforcement cost of securing user authentication process. Each row denotes the enforced policies by the administrator. For the sake of simplicity, we consider three combinations of the possible policy enforcements. These are "password", "password + CAPTCHA" and "smartcard". On the other hand, the attacker has also several strategies to compromise the authentication process. These are "scriptbot" and "scriptbot + OCR engine". These strategies are represented as columns. As we can see, this matrix resembles the *normal form game*, which is known in the field of game theory [50]. Each element of this matrix represents the loss or gain (negative or positive value) for each player (administrator or hacker). Thus, each player should choose a row or a column, which results in the least loss or most gain for each player. Intuitively, the most rational choice for both players is to choose the second strategy (password + CAPTCHA and scriptbot + OCR engine). This choice is also known as *Nash equilibrium* [67].

- *Application of hierarchical finite state machines to define fine-granular abstraction levels of security policies*
  The presented classification of security policies is too simple for the real-world scenario. This definition serves only as a general categorization of the security policies.

  The abstract policy class, which is one of the abstraction levels of workflow policies, can be further divided into several sub levels. The number of layer division depends on the organizational processes or workflows, in which the security policies apply.

  Based on this fact, the height of the abstraction level of the security policies strongly depends on the detail of the considered workflow. However, specifying the workflow in more detail, from organizational level down to implementation architecture level, is very impracticable.

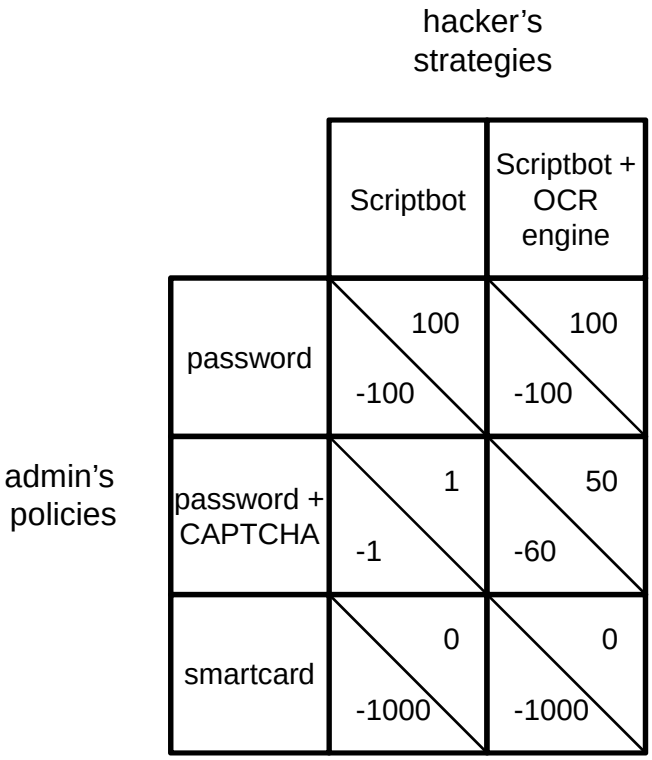

Figure 8.1: The cost matrix of a policy enforcement

To solve this problem, one may also adapt the hierarchical automata [62] or hierarchical finite state machines [44, 5] to facilitate the hierarchical representation of the workflows. The hierarchical representation allows the specification of a workflow in several abstraction level. This is useful when the workflow engineer or system analyst wants to collaborately and independently specify a workflow at the organizational level or at the system architecture level, respectively. Thus, the specification of workflow in more detail is then practicable.

- *Verification of the Refined Low-Level Policies*
  The set of refined low-level policies generated from this refinement process only denotes that some security mechanisms should be enforced within particular traces of the Kripke model, which represents the behavior of the system.

  According to expert knowledge, these assertions shall exclude any occurrence of malicious traces in the Kripke model. However, the refined policies only assert that they will exclude any malicious traces, which are known to security experts. Furthermore, these policies are only derived from best-practice solutions.

  The policy verification process can be seen as a complement to this work. The verification process analyzes the enforcement process of the refined low-level security policies within the considered system and proves, whether the low-level security policies satisfy the high-level security policies.

Bibliography

[1] Mohammed Naoufal Adraoui. Anwendung von Semantic Wiki für die Gewinnung von Expertenwissen. Diplomarbeit, Technische Universität Darmstadt, July 2007.

[2] M. Ahmed, A. Anjomshoaa, T.M. Nguyen, and A.M. Tjoa. Towards an Ontology-based Risk Assessment in Collaborative Environment Using the SemanticLIFE. In *Proceedings of the The Second International Conference on Availability, Reliability and Security*, pages 400–407. IEEE Computer Society Washington, DC, USA, 2007.

[3] M. Ahmed, H.H. Hoang, M.S. Karim, S. Khusro, M. Lanzenberger, K. Latif, E. Michlmayr, K. Mustofa, H.T. Nguyen, and A. Rauber. SemanticLIFE-a framework for managing information of a human lifetime. In *Proceedings of the 6th International Conference on Information Integration and Web-based Applications and Services*, 2004.

[4] C. Alexander et al. *The timeless way of building*. Oxford University Press, 1979.

[5] R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. *Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, pages 169–178, 1999.

[6] F. Baader. Terminological cycles in a description logic with existential restrictions. *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 02–02, 2003.

[7] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[8] F. Baader, C. Lutz, M. Milicic, U. Sattler, and F. Wolter. Integrating Description Logics and Action Formalisms: First Results. *Proceedings*

of the 20th National Conference on Artificial Intelligence (AAAI-05), 2005.

[9] T. Ball and S.K. Rajamani. *Automatically validating temporal safety properties of interfaces*, volume 2057 of *Lecture Notes in Computer Science*, page 103. Springer Verlag, 2001.

[10] A. K. Bandara, E. C. Lupu, J. Moffett, and A. Russo. A goal-based approach to policy refinement. *Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004 (POLICY 2004)*, pages 229–239, 2004.

[11] R. E. Barlow and H. E. Lambert. Introduction to Fault Tree Analysis. *Society for Industrial and Applied Mathematics*, 1975.

[12] K. Beck and W. Cunningham. Using pattern languages for object-oriented programs. *Specification and Design for Object-Oriented Programming (OOPSLA-87)*, 67, 1987.

[13] Shoham Ben-David, Richard Trefler, and Grant Weddell. Model checking the basic modalities of ctl with description logic. In Bijan Parsia, Ulrike Sattler, and David Toman, editors, *Proceedings of the 2006 International Workshop on Description Logics*, volume 189. CEUR-WS.ORG, 2006.

[14] W. Bibel, Hölldober, and Schaub. *Wissensrepräsentation und Inferenz*. Vieweg Verlag, 1998.

[15] Matt Bishop. *Computer Security: Art and Science*. Addison Wesley, 2003.

[16] Joachim Biskup and Barbara Sprick. *Towards Unifying Semantic Constraints and Security Constraints*, volume 2582 of *Lecture Notes in Computer Science*, pages 34–62. Springer Verlag, 2003.

[17] G. Booch, J. Rumbaugh, and I. Jacobson. *The unified modeling language user guide*. Addison-Wesley, 1999.

[18] Jan O. Borchers. A pattern approach to interaction design. *AI & Society*, 15(4):359–376, 2001.

[19] BSI. GSTOOL – Das BSI Tool zum IT-Grundschutz. `http://www.bsi.bund.de/gstool/index.htm`.

[20] F.J. Budinsky, M.A. Finnie, J.M. Vlissides, and P.S. Yu. Automatic Code Generation from Design Patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[21] Miriam J. Maulloand Seraphin B. Calo. Policy management: an architecture and approach. *Proceedings of the IEEE First International Workshop on Systems Management, 1993*, pages 13–26, 1993.

[22] E. Clarke and S. Jha. Tree-like counterexamples in model checking. In *17th Annual IEEE Symposium on Logic in Computer Science*, pages 19–29, 2002.

[23] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[24] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

[25] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub) Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1367–1372, 2004.

[26] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, second printing edition, 1990.

[27] Jason Crampton. Specifying and enforcing constraints in role-based access control. In *Proceedings of the eighth ACM symposium on Access control models and technologies (SACMAT '03)*, pages 43–50, New York, NY, USA, 2003. ACM Press.

[28] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Selected Papers of the Sixth International Workshop on Software Specification and Design*, pages 3–50, 1993.

[29] R. Darimont. *Process Support for Requirements Elaboration*. PhD thesis, Universite catholique de Louvain, June 2003.

[30] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 179–190, 1996.

[31] F.M. Donini, M. Lenzerini, D. Nardi, W. Nutt, and A. Schaerf. An epistemic operator for description logics. *Artificial Intelligence*, 100(1):225–274, 1998.

[32] F.M. Donini, M. Lenzerini, D. Nardi, A. Schaerf, B. Nebel, C. Rich, and W. Swartout. Adding Epistemic Operators to Concept Languages. *KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, pages 342–353, 1992.

[33] Francesco M. Donini, Daniele Nardi, and Riccardo Rosati. Description logics of minimal knowledge and negation as failure. *ACM Trans. Comput. Logic*, 3(2):177–225, 2002.

[34] M. Dumas and A.H.M. ter Hofstede. UML activity diagrams as a workflow specification language. *Proc. of the 4th Int. Conference on the Unified Modeling Language (UML01)*, 2185:76–90, 2001.

[35] C. Eckert. *IT-sicherheit: Konzepte-Verfahren-Protokolle*. Oldenbourg Wissenschaftsverlag, 2006.

[36] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with Temporal Logic on Truncated Paths. In *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 27–39. Springer, 2003.

[37] R. Eshuis. Symbolic model checking of UML activity diagrams. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):1–38, 2006.

[38] R. Eshuis and R. Wieringa. *A real-time execution semantics for UML activity diagrams*, volume 2029 of *Lecture Notes in Computer Science*, pages 76–90. Springer Verlag, 2001.

[39] R. Eshuis and R. Wieringa. Verification support for workflow design with UML activity graphs. *Proceedings of the 24th international conference on Software engineering*, pages 166–176, 2002.

[40] M.S. Feather. Requirements reconnoitring at the juncture of domain and instance. *Proceedings of IEEE International Symposium on Requirements Engineering*, pages 73–76, 1993.

[41] D. Fensel, F. van Harmelen, I. Horrocks, D.L. McGuinness, and P.F. Patel-Schneider. OIL: an ontology infrastructure for the Semantic Web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[43] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. *Proceedings of the 17th international conference on Software engineering*, pages 179–185, 1995.

[44] A. Girault, B. Lee, and EA Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, 1999.

[45] D. Grigori and M. Bouzeghoub. Service retrieval based on behavioral specification. *Services Computing, 2005 IEEE International Conference on*, 1, 2005.

[46] Object Management Group. Business process modeling notation, v1.1. Technical report, Object Management Group, 2008.

[47] V. Haarslev and R. Möller. Racer: A Core Inference Engine for the Semantic Web. *2nd International Workshop on Evaluation of Ontology-based Tools (EON-2003)*, 2003.

[48] N. Harrison. The Language of Shepherds: A Pattern Language for Shepherding. *Proceedings of the 6 thAnnual Conference on the Pattern Languages of Programs*, pages 15–18, 1999.

[49] J. Hendler and D.L. McGuinness. The DARPA Agent Markup Language. *IEEE Intelligent Systems*, 15(6):67–73, 2000.

[50] M.J. Holler and G. Illing. Einführung in die Spieltheorie, 5. überarb. *Aufl., Berlin*, 2003.

[51] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *Logic Journal of IGPL*, 8(3):239, 2000.

[52] G. E. Hughes and M. J. Cresswell. *A New Introduction to Modal Logic*. Routledge, 1996.

[53] International Organization for Standardization. BS ISO/IEC 27002:2005 Information technology. Code of practice for information security management. Technical report, 2005.

[54] H. Kaiya, H. Horai, and M. Saeki. AGORA: attributed goal-oriented requirements analysis method. *IEEE Joint International Conference on Requirements Engineering*, pages 13–22, 2002.

[55] N. L. Kerth and W. Cunningham. Using Patterns to Improve Our Architectural Vision. *IEEE Software*, 14(1):53–59, 1997.

[56] S. Kripke. Semantical analysis of modal logic I: Normal modal propositional calculi. *Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[57] E.C. Lupu, D.A. Marriott, M.S. Sloman, and N. Yialelis. A policy based role framework for access control. *Proceedings of the first ACM Workshop on Role-based access control*, 1996.

[58] B. Mahleko. *Efficient Matchmaking of Business Processes in Web Service Infrastructures*. PhD thesis, TU Darmstadt, 2006.

[59] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag New York Inc., 1995.

[60] W. Menzel and P. H. Schmitt. *Formale Systeme*. Lecture Notes of Universität Karlsruhe, 2001.

[61] B.T. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(5):493–504, 1998.

[62] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical Automata as Model for Statecharts. *Proceedings of the Third Asian Computing Science Conference on Advances in Computing Science*, pages 181–196, 1997.

[63] R. Miller and M. Shanahan. The Event Calculus in Classical Logic-Alternative Axiomatisations. *Linkoping Electronic Articles in Computer and Information Science*, 4(16):1–27, 1999.

[64] JD Moffett and MS Sloman. Policy hierarchies for distributed systems management. *IEEE Journal on Selected Areas in Communications*, 11(9):1404–1414, 1993.

[65] M.C. Mont, A. Baldwin, and C. Goh. POWER Prototype: Towards Integrated Policy-Based Management. *HP Laboratories Bristol, Bristol, UK October*, 1999.

[66] M.A. Musen, R.W. Fergerson, W.E. Grosso, N.F. Noy, M. Crubezy, and J.H. Gennari. Component-Based Support for Building Knowledge-Acquisition Systems. *Conference on Intelligent Information Processing (IIP 2000) of the International Federation for Information Processing World Computer Congress (WCC 2000)*, 194, 2000.

[67] John Forbes Nash. Non-cooperative games. *Annals of Mathematics*, 54(2):286–295, 1951.

[68] B. Nebel. Terminological cycles: Semantics and computational properties. *Principles of Semantic Networks*, pages 331–361, 1991.

[69] Gustaf Neumann and Mark Strembeck. A scenario-driven role engineering process for functional RBAC roles. In *Proceedings of the seventh ACM symposium on Access control models and technologies (SACMAT '02)*, pages 33–42, New York, NY, USA, 2002. ACM Press.

[70] Gustaf Neumann and Mark Strembeck. An approach to engineer and enforce context constraints in an RBAC environment. In *Proceedings of the eighth ACM symposium on Access control models and technologies (SACMAT '03)*, pages 65–79, New York, NY, USA, 2003. ACM Press.

[71] OASIS. eXtensible Access Control Markup Language (XACML) Version 2.0. Technical Report 6, OASIS, 2005.

[72] OASIS. Web services business process execution language version 2.0. Technical report, OASIS, 2007.

[73] M. Paolucci, T. Kawamura, T.R. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. *Proceedings of the 1st International Semantic Web Conference (ISWC '02)*, 348, 2002.

[74] B. Parsia and E. Sirin. Pellet: An OWL DL Reasoner. *Proceedings of the International Workshop on Description Logics*, 2004.

[75] T.R. Peltier. *Information Security Policies, Procedures, and Standards: Guidelines for Effective Information Security Management.* Auerbach Publications, 2002.

[76] C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Rheinisch-Westfälisches Institut fü instrumentelle Mathematik, 1962.

[77] M.R. Quillian. *Semantic Memory.* Air Force Cambridge Research Laboratories, Office of Aerospace Research, United States Air Force, 1966.

[78] Taufiq Rochaeli and Claudia Eckert. RBAC Policy Engineering with Patterns. In L. Kagal, T. Finin, and J. Hendler, editors, *Proceedings of the Semantic Web and Policy Workshop*, Galway, Ireland, November 7 2005.

[79] Taufiq Rochaeli and Claudia Eckert. Expertise Knowledge-Based Policy Refinement Process. *Proceedings of the 8th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'07)*, 2007.

[80] Taufiq Rochaeli and Claudia Eckert. Model Checking of Restricted CTL* formulas using ALCKR+. *2007 International Workshop on Description Logics (DL2007)*, 2007.

[81] Taufiq Rochaeli and Claudia Eckert. Using patterns paradigm to refine workflow policies. *1st International Workshop on Secure systems methodologies using patterns (SPattern'07)*, 2007.

[82] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, G. Pavlou, and A.L. Lafuente. Using Linear Temporal Model Checking for Goal-Oriented Policy Refinement Frameworks. *POLICY '05*, pages 181–190, 2005.

[83] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Communications of the ACM 17,7*, 1974.

[84] Ulrike Sattler. Description logic reasoners. online, 2007. http://www.cs.man.ac.uk/ sattler/reasoners.html.

[85] A. Schaad, V. Lotz, and K. Sohr. A model-checking approach to analysing organisational controls in a loan origination process. In *Proceedings of the eleventh ACM symposium on Access control models and technologies (SACMAT '06)*, pages 139–149. ACM Press New York, NY, USA, 2006.

[86] A. Schaad and J. Moffett. Separation, review and supervision controls in the context of a credit application process: a case study of organisational control principles. *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1380–1384, 2004.

[87] S. Schaffert, A. Gruber, and R. Westenthaler. A Semantic Wiki for Collaborative Knowledge Formation. *Proceedings of SEMANTICS 2005*, pages 23–25, 2005.

[88] Klaus Schild. Combining terminological logics with tense logic. In *EPIA '93: Proceedings of the 6th Portuguese Conference on Artificial Intelligence*, pages 105–120, London, UK, 1993. Springer-Verlag.

[89] Albrecht Schmiedel. A temporal terminological logic. Technical report, Technische Universität Berlin, Berlin, Germany, Germany, 1990.

[90] B. Schneier. Attack Trees. *Dr. Dobbs Journal*, 24(12):21–29, 1999.

[91] M. Schumacher. *Security engineering with patterns*. Springer New York, 2003.

[92] M. Sloman and E. Lupu. Policy Specification for Programmable Networks. *Proceedings of IWAN*, pages 73–84, 1999.

[93] A. Souzis. Building a Semantic Wiki. *IEEE Intelligent Systems*, 20(5):87–91, 2005.

[94] Stanford Center for Biomedical Informatics Research. Protege-OWL API. `http://protege.stanford.edu/plugins/owl/api/index.html`.

[95] Mark Strembeck and Gustaf Neumann. An integrated approach to engineer and enforce context constraints in RBAC environments. *ACM Transactions on Information and System Security*, 7(3):392–427, 2004.

[96] B. Suntisrivaraporn. Optimization and Implementation of Subsumption Algorithms for The Description Logic EL with Cyclic TBoxes and General Concept Inclusion Axioms. Master's thesis, Technische Universität Dresden, 2004.

[97] AG Sutcliffe and N.A.M. Maiden. Bridging the requirements gap: policies, goals and domains. *Proceedings of the Seventh International Workshop on Software Specification and Design*, pages 52–55, 1993.

[98] R. Tazzoli, P. Castagna, and S.E. Campanini. Towards a semantic wiki wiki web. *Proceedings of the International Semantic Web Conference (ISWC '04)*, 2004.

[99] The Apache Software Foundation. Xerces. `http://xerces.apache.org/`.

[100] W.M.P. van der Aalst, A.P. Barros, A.H.M. ter Hofstede, and B. Kiepuszewski. Advanced Workflow Patterns. *7th International Conference of Cooperative Information Systems (CoopIS '00)*, 2000.

[101] Axel van Lamsweerde. *Building Formal Requirements Models for Reliable Software*, volume 2043 of *Lecture Notes in Computer Science*. Springer Verlag, January 2001.

[102] Axel van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, 2001.

[103] Axel van Lamsweerde. *From System Goals to Software Architecture*, volume 2804 of *Lecture Notes in Computer Science*, pages 25–43. Springer Verlag, November 2003.

[104] Axel van Lamsweerde. Elaborating security requirements by construction of intentional anti-models. In *26th International Conference on Software Engineering(ICSE '04)*, pages 148–157, 2004.

[105] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D.F. Hasl. *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, 1981.

[106] R. Wies. Using a Classification of Management Policies for Policy Specification and Policy Transformation. *Integrated Network Management IV*, 4:44–56, 1995.

[107] C. Wolter and A. Schaad. *Modeling of Task-Based Authorization Constraints in BPMN*, volume 4714 of *Lecture Notes In Computer Science*, page 64. Springer, 2007.

[108] F. Wolter and M. Zakharyaschev. Temporalizing Description Logics. *Frontiers of Combining Systems*, 2:377–401, 2000.

[109] A. Wombacher, P. Fankhauser, B. Mahleko, and E. Neuhold. Matchmaking for business processes based on choreographies. *IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE '04)*, pages 359–368, 2004.

[110] Worfklow Management Coalition. "XPDL" Support & Resources. `http://www.wfmc.org/standards/xpdl.htm`.

[111] E. S. K. Yu. Modeling Organizations for Information Systems Requirements Engineering. *Proceedings of IEEE International Symposium on Requirements Engineering*, pages 34–41, 1993.

Appendix A

Proofs of the CTL* Semantics

To construct our proof presented in Figure 4.6, we shall consider the epistemic operator in description logic presented by Donini et al. [31]. The introduction of **K** operator allows us to denote the membership of an individual to a concept, which is explicitly asserted by the corresponding TBox. In other words, the concept $\mathbf{K}C$ denotes a set of instances, which are *known* by the knowledge base as the members of the concept C [31, p. 11]. This understanding plays a significant role in constructing the proof, as we shall see later.

This chapter begins with the introduction of the Kripke model and its representation in DL-based knowledge base. Section A.4 presents the construction of the CTL* semantics on top of the Description Logic semantics including their proofs.

A.1 Description Logics Knowledge Base

A description logic knowledge base consists of two compartments and a reasoning engine. These two compartments are called *terminology box* (TBox) and *assertion box* (ABox).

The TBox contains the definitions of terminologies and their relationships to other terminologies. In other words, the TBox contains the *ontology*. In description logics, these terminologies are known as *concepts*. A concept can be defined as more general than or equivalent to another concept. It can also have several semantic relationships to another concepts. These semantic relationships are known as *roles*. The TBox is constructed by the description logic language, which will be explained below.

The ABox contains the assertions, which state the membership of an entity to a particular concept. In description logics, an entity is known as

individual. Such assertions are usually written as $\mathsf{C}(a)$, where C and a are the concept and the individual, respectively. The ABox also contains the assertions stating the binary relationships between the individuals. These assertions are written as $\mathsf{R}(a, b)$, whereas R is the role name and a and b are the individuals.

A.1.1 Description Logics Language

There are several definitions of DL language, which are used to construct the TBox. The most general and simple language is the $\mathcal{AL}$ language, which is known as "attributive language". Other languages are defined by adding new extensions this base language. For example, the $\mathcal{ALC}$ language is the $\mathcal{AL}$ language added with the negation.

In our study we consider the $\mathcal{SHK}$ language. This language is based on $\mathcal{ALC}$ and extended by adding the: (i) role transitivity, (ii) role hierarchy and (iii) **K** epistemic operator.

Thus, every $\mathcal{SHK}$ concept can be defined by the following syntax:

$$\mathcal{SHK} ::= \bot \mid \top \mid A \mid \neg C \mid \mathbf{K}C \mid C \sqcup D \mid C \sqcap D \mid \exists R.C \mid \forall R.C.$$

A is the name of an atomic concept and C and D are the names of the complex concepts, which are recursively defined by this syntax. R is the role name.

A.1.2 Formal Semantics of $\mathcal{SHK}$

The formal semantics of description logics is based on the interpretation $\mathcal{I}$, which consists of a non-empty set Δ and interpretation functions that assign every atomic concept A to a set $A^{\mathcal{I}} \subseteq \Delta$ and every atomic role R to a binary relation $R^{\mathcal{I}} \subseteq \Delta \times \Delta$. Furthermore, the interpretation functions also assign an individual a to $a^{\mathcal{I}} \in \Delta$. Figure A.1 shows the formal semantics of $\mathcal{SHK}$.

A.1.3 Rigid Term Assumption

An important property of the interpretation of DL semantics in presence of epistemic operator is the rigid term assumption. Donini et al. state in [32] that for all interpretations, every individual is assigned to the same element of the domain interpretation Δ. This means that all interpretations have fixed assignments from each individual to a fixed element of interpretation domain Δ. Formally, we have $\forall \mathcal{I}, \mathcal{J} \in \mathcal{W} : x^{\mathcal{I}} = x^{\mathcal{J}}$.

$$
\begin{aligned}
\top^{\mathcal{I}} &= \Delta \\
\bot^{\mathcal{I}} &= \emptyset \\
A^{\mathcal{I}}, \mathcal{W} &= A^{\mathcal{I}} \\
R^{\mathcal{I}}, \mathcal{W} &= R^{\mathcal{I}} \\
(\neg A)^{\mathcal{I},\mathcal{W}} &= \Delta \setminus A^{\mathcal{I},\mathcal{W}} \\
(C \sqcap D)^{\mathcal{I},\mathcal{W}} &= C^{\mathcal{I},\mathcal{W}} \cap D^{\mathcal{I},\mathcal{W}} \\
(C \sqcup D)^{\mathcal{I},\mathcal{W}} &= C^{\mathcal{I},\mathcal{W}} \cup D^{\mathcal{I},\mathcal{W}} \\
(\forall R.C)^{\mathcal{I},\mathcal{W}} &= \{a \in \Delta \mid \forall b : (a,b) \in R^{\mathcal{I},\mathcal{W}} \rightarrow b \in C^{\mathcal{I},\mathcal{W}}\} \\
(\exists R.C)^{\mathcal{I},\mathcal{W}} &= \{a \in \Delta \mid \exists b : (a,b) \in R^{\mathcal{I},\mathcal{W}} \wedge b \in C^{\mathcal{I},\mathcal{W}}\} \\
(\mathbf{K}C)^{\mathcal{I},\mathcal{W}} &= \bigcap_{\mathcal{J} \in \mathcal{W}} C^{\mathcal{J},\mathcal{W}} = \{a \in \Delta \mid \forall \mathcal{J} \in \mathcal{W} : a \in C^{\mathcal{I},\mathcal{W}}\} \\
(\mathbf{K}R)^{\mathcal{I},\mathcal{W}} &= \bigcap_{\mathcal{J} \in \mathcal{W}} R^{\mathcal{J},\mathcal{W}} = \{a \in \Delta \mid \forall \mathcal{J} \in \mathcal{W} : (a,b) \in R^{\mathcal{I},\mathcal{W}}\}
\end{aligned}
$$

Figure A.1: Formal semantics of $\mathcal{SHK}$

A.2 Kripke Model

A.2.1 Representing Kripke Model in Description Logic

First, we present the notations used in Kripke model as follows:

$K:$	$\langle W, I, RL, L \rangle$
$W:$	a set of states
$I \subset W:$	a set of initial states
$RL \subseteq W \times W:$	a set of relations between states
$L:$	$W \rightarrow 2^{AP}$, a function that maps a state to a set of atomic propositions
$AP:$	a set of atomic propositions
$p_i \in AP:$	an atomic proposition
$s:$	state
$\pi:$	a sequence of states in Kripke model
$\pi_i:$	the state at i-th position in path π, $i \geq 0$

To represent the Kripke model in DL-based knowledge base, we define the $\mathcal{SHK}$ knowledge base $\mathcal{KB} : \langle \mathcal{T}_{\mathcal{KB}}, \mathcal{A}_{\mathcal{KB}} \rangle$. It consists of a terminology box $\mathcal{T}_{\mathcal{KB}}$ and an assertion box $\mathcal{A}_{\mathcal{KB}}$.

Before we define $\mathcal{T}_{\mathcal{KB}}$ and $\mathcal{A}_{\mathcal{KB}}$, we define the following sets:

- $X = \{x | x \text{ is an individual of the assertion box}\}$
- $G = \{V | V \text{ is a concept of the terminology box}\}$

and the following functions:

- $t_1 : W \rightarrow X$ maps each state $s \in W$ into a unique individual $x \in X$.
- $t_2 : RL \rightarrow U \subset X \times X, (a, b) \mapsto (t_1(a), t_1(b))$.
- $t_3 : AP \rightarrow \mathcal{T}_{AP}$ maps each atomic proposition $p \in AP$ into a unique concept $V \in \mathcal{T}_{AP}$.
- $t_4 : W \rightarrow 2^{AP} \times W, a \mapsto (L(a), a)$.
- $t_5 : 2^{AP} \rightarrow 2^{\mathcal{T}_{AP}}, a \mapsto \{b \in \mathcal{T}_{AP} | \forall c \in a : t_3(c) = b\}$.
- $t_6 : 2^{AP} \times W \rightarrow 2^{\mathcal{T}_{AP}} \times X, (a, b) \mapsto (t_5(a), t_1(b))$.
- $t_7 : 2^{\mathcal{T}_{AP}} \times X \rightarrow 2^{\mathcal{T}_{AP} \times X}$,
  $(a, b) \mapsto \{(c, d) \in \mathcal{T}_{AP} \times X | \forall e \in \mathcal{T}_{AP} : (e \in a \rightarrow c = e \wedge d = b)\}$.
- $t_8 : W \rightarrow 2^{\mathcal{T}_{AP} \times X}$,
  $a \mapsto \{(b, c) \in 2^{\mathcal{T}_{AP} \times X} | (b, c) \in t_7(t_6(t_4(a)))\}$
- $t_9 : F \rightarrow \mathcal{T}_f, f_i \mapsto D_i$

The Kripke model is represented in the knowledge base by performing these mappings:

- Mapping the states into individuals
  Each state $s \in W$ is uniquely mapped into an individual $x \in X$. This process is performed by using the function t_1.
- Mapping the labels into concepts
  Each label $p \in AP$ is mapped into a concept $V \in \mathcal{T}_{AP}$ that is performed by using the function t_2.
- Mapping the state transitions into ABox assertions
  Each tupel $(s_1, s_2) \in RL$ is mapped into a tupel $(x_1, x_2) \in U$, which is performed by using the function t_3. This tupel is used to generate an ABox assertion $\mathit{next}(x_1, x_2)$, which is stored in the ABox $\mathcal{A}_L$.

- Mapping the state labels into ABox assertions
  We perform the transformation by using the mapping function t_8 that maps each state $s \in W$ into a set of tupels $(p, x) \in 2^{\mathcal{T}_{AP} \times X}$. More specifically, the function t_8 performs the following mapping: $W \rightarrow 2^{AP} \times W \rightarrow 2^{\mathcal{T}_{AP}} \times X \rightarrow 2^{\mathcal{T}_{AP} \times X}$. Each tupel is used to costruct an ABox assertion. This assertion is then stored in the ABox $\mathcal{A}_R$. Each assertion represents a label, which is given to a certain state. For example, the tupel $(\mathsf{Label1}, x_1)$ generates the assertion $\mathsf{Label1}(x_1)$. This assertion states that the state represented by the individual x_1 has the label, which is represented by the concept $\mathsf{Label1}$.

From Section 4.4.3, we know that $\mathcal{T}_f$ contains the concepts of the translated restricted CTL* formulas. The translation is performed by the function t_9. Furthermore, we define an additional role definition $\mathit{future} \equiv \mathit{next}$. The role future is a transitive role ($\mathit{future} \in \mathbf{R}_+$).

Thus, we construct the TBox $\mathcal{T}_{\mathcal{KB}} = \mathcal{T}_{AP} \cup \mathcal{T}_{ONT} \cup \mathcal{T}_f$ and the ABox $\mathcal{A}_{\mathcal{KB}} = \mathcal{A}_L \cup \mathcal{A}_{RL}$.

In summary, we present the notations used to construct Kripke model in description logic knowledge base as follows:

x :	a unique name of individual in Abox representing a state of the Kripke model
σ :	a sequence of ABox instances which is connected by transition relationship $\mathit{next}(x_i, x_j)$
σ_i :	the individual at the i-th position in sequence σ representing the state π_i, $i \geq 0$
V_i :	an atomic concept representing a propositional atom
D :	A concept of TBox representing the restricted CTL* formula
$\mathcal{T}_{AP}$:	a set of concepts representing the atomic propositions
$\mathcal{T}_{ONT}$:	a set of definitions representing the generalization or equivalence between two atomic propositions
$\mathcal{T}_f$:	a set of concepts representing the restricted CTL* formulas
$\mathcal{T}_{\mathcal{KB}}$:	$\mathcal{T}_{AP} \cup \mathcal{T}_{ONT} \cup \mathcal{T}_f$
$\mathcal{A}_{RL}$:	a set of assertions about relationships between states
$\mathcal{A}_L$:	a set of assertions about states labeling
$\mathcal{A}_{\mathcal{KB}}$:	ABox for model checking the Kripke model $\mathcal{A}_{\mathcal{KB}} = \mathcal{A}_L \cup \mathcal{A}_{RL}$

$$\mathcal{KB}: \quad \langle \mathcal{T}_{\mathcal{KB}}, \mathcal{A}_{\mathcal{KB}} \rangle \text{, The DL knowledge base}$$

A.3 CTL* Semantics

This section presents the semantics of CTL* that serve the purpose of a reference for the proof, since we want to construct the CTL* semantics on top of the Description Logic semantics. The semantics is presented in Figure A.2.

$$\begin{aligned}
s \vDash p_i &\Leftrightarrow p_i \in L(s) \\
s \vDash \neg f_1 &\Leftrightarrow s \nvDash f_1 \\
s \vDash f_1 \vee f_2 &\Leftrightarrow s \vDash f_1 \vee s \vDash f_2 \\
s \vDash f_1 \wedge f_2 &\Leftrightarrow s \vDash f_1 \wedge s \vDash f_2 \\
s \vDash \mathbf{E}\ (g_1) &\Leftrightarrow \text{ there is a path } \pi \text{ starting with } s \text{ such that } \pi \vDash g_1 \\
\pi \vDash f_1 &\Leftrightarrow \pi_0 \vDash f_1 \\
\pi \vDash \neg g_1 &\Leftrightarrow \pi \nvDash g_1 \\
\pi \vDash g_1 \vee g_2 &\Leftrightarrow \pi \vDash g_1 \vee \pi \vDash g_2 \\
\pi \vDash g_1 \wedge g_2 &\Leftrightarrow \pi \vDash g_1 \wedge \pi \vDash g_2 \\
\pi \vDash \mathbf{X}\ g_1 &\Leftrightarrow \pi_1 \vDash g_1 \\
\pi \vDash \mathbf{G}\ g_1 &\Leftrightarrow \forall i \in \mathbb{N}_0 : \pi_i \vDash g_1 \\
\pi \vDash \mathbf{F}\ g_1 &\Leftrightarrow \exists i \in \mathbb{N}_0 : \pi_i \vDash g_1 \\
\pi \vDash g_1 \mathbf{U} g_2 &\Leftrightarrow \exists j \in \mathbb{N}_0 : \pi_j \vDash g_2 \wedge \forall i \in \{0, \ldots, j-1\} : \pi_i \vDash g_1
\end{aligned}$$

Figure A.2: CTL* semantics

A.4 CTL* Semantics in Description Logic

We present the construction of the subset of CTL* semantics in Description Logic semantics. The CTL* semantics is shown in Figure A.3. Since we deal only with the restricted CTL* formula, which has the form of $\mathbf{E}\ [\alpha]$ where $\mathbf{E}$ is the existential path quantifier and α is LTL formula. The LTL formula α should be written in the form of negation-normal-form by pushing the negation ($\neg$) inwards to atomic propositions.

$$
\begin{aligned}
s \vDash p_i &\Leftrightarrow \mathcal{KB} \vDash \mathbf{K}V_i(x) && \text{(A.1)}\\
s \vDash \neg p_i &\Leftrightarrow \mathcal{KB} \vDash \neg\mathbf{K}V_i(x) && \text{(A.2)}\\
s \vDash \mathbf{E}\ (g_1) &\Leftrightarrow \mathcal{KB} \vDash D_1(x) && \text{(A.3)}\\
\pi \vDash g_1 &\Leftrightarrow \mathcal{KB} \vDash D_1(\sigma_0) && \text{(A.4)}\\
\pi \vDash g_1 \vee g_2 &\Leftrightarrow \mathcal{KB} \vDash (D_1 \sqcup D_2)(\sigma_0) && \text{(A.5)}\\
\pi \vDash g_1 \wedge g_2 &\Leftrightarrow \mathcal{KB} \vDash (D_1 \sqcap D_2)(\sigma_0) && \text{(A.6)}\\
\pi \vDash \mathbf{X}\ g_1 &\Leftrightarrow \mathcal{KB} \vDash (\exists\mathbf{K}\mathit{next}.D_1)(\sigma_0) && \text{(A.7)}\\
\pi \vDash \mathbf{G}\ g_1 &\Leftrightarrow \langle \mathcal{T}_{\mathcal{KB}} \cup \{D_{aux} \equiv D_1 \sqcap \exists\mathbf{K}\mathit{next}.D_{aux}\}, \mathcal{A}_{\mathcal{KB}}\rangle && \\
&\quad \vDash D_{aux}\ (\sigma_0) && \text{(A.8)}\\
\pi \vDash \mathbf{F}\ g_1 &\Leftrightarrow \mathcal{KB} \vDash (D_1 \sqcup \exists\mathit{future}.D_1)\ (\sigma_0) && \text{(A.9)}\\
\pi \vDash g_1\mathbf{U}g_2 &\Leftrightarrow \langle \mathcal{T}_{KB} \cup \{D_{aux} \equiv D_2 \sqcup (D_1 \sqcap \exists\mathbf{K}\mathit{next}.D_{aux} \sqcap \exists\mathit{future}.D_2)\} && \\
&\quad , \mathcal{A}_{\mathcal{KB}}\rangle \vDash D_{aux}\ (\sigma_0) && \text{(A.10)}
\end{aligned}
$$

Figure A.3: Formal semantics of a subset of CTL* logic in $\mathcal{SHK}$

The semantics presented in Figure A.3 shows the CTL semantics, which are emulated on top of the description logic semantics. It should be noted that we define an assertion $\mathit{future} \equiv \mathit{next}$, which is used in Equation A.9.

A.4.1 Proofs

Before we present the proofs, we will show the lemmas, which support the proofs.

Lemma 1 *Let $\mathcal{KB}$ be consistent and $\mathcal{W}$ be a set of interpretations satisfying $\mathcal{KB}$.*

$$\mathcal{KB} \supset \{V_i(x)\} \Leftrightarrow \mathcal{KB} \vDash \mathbf{K}V_i\ (x).$$

Proof 4

" $\Rightarrow$ " : *Suppose that* $\mathcal{KB} \nvDash \mathbf{K}V_i\ (x)$

$\Rightarrow$ $\exists \mathcal{I} \in \mathcal{W} : x^{\mathcal{I}} \notin \bigcap_{\mathcal{J} \in \mathcal{W}} V_i^{\mathcal{J},\mathcal{W}}$

$\Rightarrow$ $\exists \mathcal{I} \in \mathcal{W}\ ,\ \exists \mathcal{J} \in \mathcal{W} : x^{\mathcal{I}} \notin V_i^{\mathcal{J}}$

$\Rightarrow$ *due to the rigid term assumption, we have* $\exists \mathcal{I}, \mathcal{J} \in \mathcal{W} : x^{\mathcal{J}} \notin V_i^{\mathcal{J}}$.

$\Rightarrow$ $\exists \mathcal{J} \in \mathcal{W} : x^{\mathcal{J}} \notin V_i^{\mathcal{J}}$

However, it contradicts with $\mathcal{KB} \supset \{V_i(x)\}$, *since*
$\mathcal{KB} \supset \{V_i(x)\} \Leftrightarrow \forall \mathcal{I} \in \mathcal{W} : x^{\mathcal{I}} \in V_i^{\mathcal{I}}$.

" $\Leftarrow$ " : *Suppose that* $\mathcal{KB} \cap \{V_i(x)\} = \emptyset$.

$\Rightarrow$ *Since there's no assertion* $V_i(x)$ *exists, it implies*
$\exists \mathcal{I}, \mathcal{J} \in \mathcal{W}\ : x^{\mathcal{I}} \notin V_i^{\mathcal{I}} \wedge x^{\mathcal{J}} \in V_i^{\mathcal{J}}$.
This is due to the open-world semantics of the description logics.

$\Rightarrow$ $\exists \mathcal{I} : x^{\mathcal{I}} \notin V_i^{\mathcal{I}}$. *However, this contradicts with* $\mathcal{KB} \vDash \mathbf{K}V_i(x)$, *since*
$\mathcal{KB} \vDash \mathbf{K}V_i(x) \Leftrightarrow \forall \mathcal{I} \in \mathcal{W} : x^{\mathcal{I}} \in \bigcap_{\mathcal{J} \in \mathcal{W}} V_i^{\mathcal{J},\mathcal{W}}$
$\Leftrightarrow \forall \mathcal{I}, \mathcal{J} \in \mathcal{W} : x^{\mathcal{I}} \in V_i^{\mathcal{J}}$
$\Leftrightarrow$ *due to the rigid term assumption,*$\forall \mathcal{J} : x^{\mathcal{J}} \in V_i^{\mathcal{J}}$.

□

Lemma 2 *Let* $\mathcal{KB}$ *be consistent and* $\mathcal{W}$ *be a set of interpretations satisfying* $\mathcal{KB}$.

$$\mathcal{KB} \cap \{V_i(x)\} = \emptyset \Leftrightarrow \mathcal{KB} \vDash \neg \mathbf{K}V_i\ (x).$$

Proof 5

" $\Rightarrow$ " : *Suppose that* $\mathcal{KB} \nvDash \neg\mathbf{K}V_i(x)$.

$\Rightarrow$ $\exists\mathcal{I} \in \mathcal{W} : x^{\mathcal{I}} \notin \Delta \setminus \bigcap_{\mathcal{J}\in\mathcal{W}} V_i^{\mathcal{J},\mathcal{W}}$

$\Rightarrow$ $\exists\mathcal{I} \in \mathcal{W}, \forall\mathcal{J} \in \mathcal{W} : x^{\mathcal{I}} \in V_i^{\mathcal{J}}$

$\Rightarrow$ *Due to the rigid term assumption,* $\forall\mathcal{J} \in \mathcal{W} : x^{\mathcal{J}} \in V_i^{\mathcal{J}}$
However, it contradicts with
$\mathcal{KB} \cap \{V_i(x)\} = \emptyset \Leftrightarrow \exists\mathcal{I} \in \mathcal{W} : x^{\mathcal{I}} \in V_i^{\mathcal{I}} \wedge$
$\exists\mathcal{J} \in \mathcal{W} : x^{\mathcal{J}} \notin V_i^{\mathcal{J}}$.

" $\Leftarrow$ " : *suppose that* $\mathcal{KB} \supset \{V_i(x)\}$.

$\Rightarrow$ $\forall\mathcal{I} \in \mathcal{W} : x^{\mathcal{I}} \in V_i^{\mathcal{I}}$

$\Rightarrow$ *due to the rigid term assumption, we have*
$\forall\mathcal{J} \in \mathcal{W}, \forall\mathcal{I} \in \mathcal{W} : x^{\mathcal{J}} \in V_i^{\mathcal{I}}$.

$\Rightarrow$ $\forall\mathcal{J} \in \mathcal{W}, \forall\mathcal{I} \in \mathcal{W} : x^{\mathcal{J}} \in V_i^{\mathcal{I},\mathcal{W}}$

$\Rightarrow$ $\forall\mathcal{J} \in \mathcal{W} : x^{\mathcal{J}} \in \bigcap_{\mathcal{I}\in\mathcal{W}} V_i^{\mathcal{I},\mathcal{W}}$

$\Rightarrow$ $\forall\mathcal{J} \in \mathcal{W} : x^{\mathcal{J}} \notin \Delta \setminus \bigcap_{\mathcal{I}\in\mathcal{W}} V_i^{\mathcal{I},\mathcal{W}}$

$\Rightarrow$ *due to the rigid term assumption, we have*
$\exists\mathcal{J} \in \mathcal{W} : x^{\mathcal{J}} \notin \Delta \setminus \bigcap_{\mathcal{I}\in\mathcal{W}} V_i^{\mathcal{I},\mathcal{W}}$.

$\Rightarrow$ $\mathcal{KB} \nvDash \neg\mathbf{K}V_i(x)$. *This contradicts with* $\mathcal{KB} \vDash \neg\mathbf{K}V_i(x)$.

□

Lemma 3 *Let* $\mathcal{KB}$ *be consistent and* $\mathcal{W}$ *be a set of interpretations satisfying* $\mathcal{KB}$. *Furthermore,* $\mathcal{KB} \vDash D_1(x_1)$.

$$\mathcal{KB} \supset \{\mathit{next}(x_0, x_1)\} \Leftrightarrow \mathcal{KB} \vDash (\exists\mathbf{K}\mathit{next}.D_1)(x_1).$$

Proof 6

$$
\begin{array}{ll}
\text{" } \Rightarrow \text{ " :} & \mathcal{KB} \supset \{\mathit{next}(x_0, x_1)\} \\
\Rightarrow & \forall \mathcal{I} \in \mathcal{W} : (x_0^{\mathcal{I}}, x_1^{\mathcal{I}}) \in \mathit{next}^{\,\mathcal{I}} \\
\Rightarrow & \textit{due to the rigid term assumption, we have} \\
& \forall \mathcal{J}, \mathcal{I} \in \mathcal{W} : (x_0^{\mathcal{J}}, x_1^{\mathcal{J}}) \in \mathit{next}^{\,\mathcal{I}} \\
\Rightarrow & \forall \mathcal{J}, \mathcal{I} \in \mathcal{W} : (x_0^{\mathcal{J}}, x_1^{\mathcal{J}}) \in \mathit{next}^{\,\mathcal{I},\mathcal{W}} \\
\Rightarrow & \forall \mathcal{J} \in \mathcal{W} : (x_0^{\mathcal{J}}, x_1^{\mathcal{J}}) \in \bigcap_{\mathcal{I} \in \mathcal{W}} \mathit{next}^{\,\mathcal{I},\mathcal{W}} \\
\Rightarrow & \textit{since } \mathcal{KB} \vDash D_1(x_1), \textit{ we have} \\
& \forall \mathcal{J} \in \mathcal{W}, \exists b \in \Delta : ((x_0^{\mathcal{J}}, b) \in \bigcap_{\mathcal{I} \in \mathcal{W}} \mathit{next}^{\,\mathcal{I},\mathcal{W}} \wedge b \in D_1^{\mathcal{J}}) \\
\Rightarrow & \forall \mathcal{J} \in \mathcal{W}, \exists b \in \Delta : ((x_0^{\mathcal{J}}, b) \in (\mathbf{K}\mathit{next})^{\,\mathcal{J},\mathcal{W}} \wedge b \in D_1^{\mathcal{J}}) \\
\Rightarrow & \forall \mathcal{J} \in \mathcal{W} : x_0^{\mathcal{J}} \in (\exists \mathbf{K}\mathit{next}.D_1)^{\mathcal{J},\mathcal{W}} \\
\Rightarrow & \mathcal{KB} \vDash (\exists \mathbf{K}\mathit{next}.D_1)(x_0).
\end{array}
$$

$$
\begin{array}{ll}
\text{" } \Leftarrow \text{ " :} & \mathcal{KB} \vDash (\exists \mathbf{K}\mathit{next}.D_1)(x_0) \\
\Rightarrow & \forall \mathcal{I} \in \mathcal{W}, \exists b \in \Delta : (x_0^{\mathcal{I}}, b) \in \mathit{next}^{\,\mathcal{I}} \wedge b \in D_1^{\mathcal{I}} \\
\Rightarrow & \textit{since } \mathcal{KB} \vDash D_1(x_1), \textit{ we have } \forall \mathcal{I} \in \mathcal{W} : (x_0^{\mathcal{I}}, x_1^{\mathcal{I}}) \in \mathit{next}^{\,\mathcal{I}} \\
\Rightarrow & \mathcal{KB} \supset \{\mathit{next}(x_0, x_1)\}.
\end{array}
$$

□

Lemma 4 *Let $\mathcal{KB}$ be consistent and $\mathcal{W}$ be a set of interpretations satisfying $\mathcal{KB}$.*

$$\exists k \in \mathbb{N}, \exists l \leq k : \mathcal{KB} \supset \{\mathit{next}(\sigma_k, \sigma_l)\} \wedge$$

$$\forall i \in \mathbb{N} : (i \leq k \rightarrow \mathcal{KB} \vDash D_1(\sigma_i)) \wedge (i < k \rightarrow \mathcal{KB} \supset \{\mathit{next}(\sigma_i, \sigma_{i+1})\})$$

$$\Leftrightarrow$$

$$\mathcal{KB} \cup \{D_{aux} \equiv D_1 \sqcap \exists \mathbf{K}\mathit{next}.D_{aux}\} \vDash D_{aux}(\sigma_0)$$

Proof 7

" $\Rightarrow$ " :

case 1: $(k = 0)$

$\mathcal{KB} \vDash D_1(\sigma_0) \wedge \mathcal{KB} \supset \{\mathit{next}(\sigma_0, \sigma_0)\}$

$\Rightarrow$ $\forall \mathcal{I} \in \mathcal{W} : D_1^{\mathcal{I}} \supset \{\sigma_0^{\mathcal{I}}\} \wedge \mathit{next}^{\mathcal{I}} \supset \{(\sigma_0^{\mathcal{I}}, \sigma_0^{\mathcal{I}}).\}$

Clearly, $\forall \mathcal{I} \in \mathcal{W} : \sigma_0^{\mathcal{I}} \in D_{aux}^{\mathcal{I}}$,

since $\forall \mathcal{I} \in \mathcal{W} : \sigma_0^{\mathcal{I}} \in D_1^{\mathcal{I}}$ *and*

$\forall \mathcal{I} \in \mathcal{W} : (\sigma_0^{\mathcal{I}}, \sigma_0^{\mathcal{I}}) \in \mathit{next}^{\mathcal{I}} \overset{\text{rigid term assumption}}{\Leftrightarrow}$

$\forall \mathcal{I} \in \mathcal{W} : (\sigma_0^{\mathcal{I}}, \sigma_0^{\mathcal{I}}) \in \bigcap_{\mathcal{J} \in \mathcal{W}} \mathit{next}^{\mathcal{J}}.$

case 2: $(k > 0)$

$\exists k > 0, \exists l \leq k, \forall m \in \{0, \ldots, k\} : \mathcal{KB} \vDash D_1(\sigma_m) \wedge$

$\mathcal{KB} \supset \{\mathit{next}(\sigma_0, \sigma_1), \ldots, \mathit{next}(\sigma_{k-1}, \sigma_k), \mathit{next}(\sigma_k, \sigma_l)\}$

$\Rightarrow$ $\forall \mathcal{I} \in \mathcal{W} : D_1^{\mathcal{I}} \supset \{\sigma_0^{\mathcal{I}}, \ldots, \sigma_k^{\mathcal{I}}\} \wedge$

$\mathit{next}^{\mathcal{I}} \supset \{(\sigma_0^{\mathcal{I}}, \sigma_1^{\mathcal{I}}), \ldots, (\sigma_{k-1}^{\mathcal{I}}, \sigma_k^{\mathcal{I}}), (\sigma_k^{\mathcal{I}}, \sigma_l^{\mathcal{I}})\}$

$\Rightarrow$ $\forall \mathcal{I} \in \mathcal{W}$: *there's a cycle in the* next

assertions and $\forall m \in \{0, \ldots, k\} : \sigma_m^{\mathcal{I}} \in D_1^{\mathcal{I}}$,

we have $\forall \mathcal{I} \in \mathcal{W} : \sigma_0^{\mathcal{I}} \in D_{aux}^{\mathcal{I}}$.

" $\Leftarrow$ " :

$\mathcal{KB} \cup \{D_{aux} \equiv D_1 \sqcap \exists \mathbf{K}\mathit{next}.D_{aux}\} \vDash D_{aux}(\sigma_0)$

$\Rightarrow$ $\exists k \geq 0, \exists l \leq k, \forall \mathcal{I} \in \mathcal{W} :$

$\mathit{next}^{\mathcal{I}} = \{(\sigma_0^{\mathcal{I}}, \sigma_1^{\mathcal{I}}), \ldots, (\sigma_{k-1}^{\mathcal{I}}, \sigma_k^{\mathcal{I}}), (\sigma_k^{\mathcal{I}}, \sigma_l^{\mathcal{I}})\}$

$\wedge \forall m \in \{0, \ldots, k\} : \sigma_m^{\mathcal{I}} \in D_1^{\mathcal{I}}$

$\Rightarrow$ $\exists k \geq 0, \exists l \leq k :$

$\mathcal{KB} \supset \{\mathit{next}(\sigma_0, \sigma_1), \ldots, \mathit{next}(\sigma_{k-1}, \sigma_k), \mathit{next}(\sigma_k, \sigma_l)\} \wedge$

$\forall m \in \{0, \ldots, k\} : \mathcal{KB} \vDash D_1(\sigma_m)$

$\Rightarrow$ $\exists k \in \mathbb{N}, \exists l \leq k : \mathcal{KB} \supset \{\mathit{next}(\sigma_k, \sigma_l)\} \wedge \forall i \in \mathbb{N} :$

$(i \leq k \rightarrow \mathcal{KB} \vDash D_1(\sigma_i)) \wedge (i < k \rightarrow \mathcal{KB} \supset \{\mathit{next}(\sigma_i, \sigma_{i+1})\}).$

□

Lemma 5 *Let* $\mathcal{KB}$ *be consistent and* $\mathcal{W}$ *be a set of interpretations satisfying* $\mathcal{KB}$.

$$\exists k \geq 0 : \mathcal{KB} \vDash D_1(\sigma_k) \wedge \forall l < k : \mathcal{KB} \supset \{\mathit{next}(\sigma_l, \sigma_{l+1})\}$$

$$\Leftrightarrow$$
$$\mathcal{KB} \models (D_1 \sqcup \exists \textit{future}.D_1)(\sigma_0).$$

Proof 8

" $\Rightarrow$ " :

$k = 0$:

$\mathcal{KB} \models D_1(\sigma_0)$

$\Rightarrow$ $\mathcal{KB} \models (D_1 \sqcup \exists \textit{future}.D_1)(\sigma_0)$.

$k > 0$:

$\exists k > 0, \forall l \in \{0, \ldots, k-1\} : \mathcal{KB} \supset \{\textit{next}(\sigma_l, \sigma_{l+1})\} \wedge \mathcal{KB} \models D_1(\sigma_k)$

$\Rightarrow$ $\exists k > 0, \forall \mathcal{I} \in \mathcal{W} : \bigwedge_{0 \leq l \leq k-1} (\sigma_l^{\mathcal{I}}, \sigma_{l+1}^{\mathcal{I}}) \in \textit{next}^{\mathcal{I}} \wedge$

$\sigma_k^{\mathcal{I}} \in D_1^{\mathcal{I}}$

$\Rightarrow$ *since* $\textit{future} \equiv \textit{next}$,

$\exists k > 0, \forall \mathcal{I} \in \mathcal{W} : \bigwedge_{0 \leq l \leq k-1} (\sigma_l^{\mathcal{I}}, \sigma_{l+1}^{\mathcal{I}}) \in \textit{future}^{\mathcal{I}} \wedge$

$\sigma_k^{\mathcal{I}} \in D_1^{\mathcal{I}}$

$\Rightarrow$ *since* $\textit{future}$ *is a transitive role, we have*

$\exists k > 0, \forall \mathcal{I} \in \mathcal{W} : (\sigma_0^{\mathcal{I}}, \sigma_k^{\mathcal{I}}) \in \textit{future}^{\mathcal{I}} \wedge \sigma_k^{\mathcal{I}} \in D_1^{\mathcal{I}}$

$\Rightarrow$ $\exists k > 0, \forall \mathcal{I} \in \mathcal{W}, \exists b \in \Delta : (\sigma_0^{\mathcal{I}}, b) \in \textit{future}^{\mathcal{I}} \wedge b = \sigma_k^{\mathcal{I}} \in D_1^{\mathcal{I}}$

$\Rightarrow$ $\exists k > 0 : \mathcal{KB} \models (\exists \textit{future}.D_1)(\sigma_0)$

$\Rightarrow$ $\exists k > 0 : \mathcal{KB} \models (D_1 \sqcup \exists \textit{future}.D_1)(\sigma_0)$.

" $\Leftarrow$ " :

$k = 0$:

suppose that $\mathcal{KB} \not\models D_1(\sigma_0)$

$\Rightarrow$ *since there's no sequence of role assertions* $\textit{next}$ *starting from* σ_0 *and we have* $\mathcal{KB} \not\models \exists \textit{future}.D_1(\sigma_0)$.

$\Rightarrow$ $\mathcal{KB} \models (D_1 \sqcup \exists \textit{future}.D_1)(\sigma_0)$.

$k > 0$:

Suppose that

$\forall k > 0, \underbrace{\exists l \in \{0, \ldots, k-1\} : \mathcal{KB} \cap \{\textit{next}(\sigma_l, \sigma_{l+1})\} = \emptyset}_{a} \vee$

$\underbrace{\mathcal{KB} \not\models D_1(\sigma_k)}_{b}$

case 1: *(a is true)*
$\forall k > 0, \exists l \in \{0, \ldots, k-1\} : \mathcal{KB} \cap \{\mathit{next}(\sigma_l, \sigma_{l+1})\} = \emptyset$
$\Rightarrow$ *let* $l = 0$*, thus we have*
$\forall k > 0 : \mathcal{KB} \cap \{\mathit{next}(\sigma_0, \sigma_1)\} = \emptyset$
$\forall k > 0 : \mathcal{KB} \not\models (\exists \mathit{next}.\{\sigma_1\})(\sigma_0)$
$\Rightarrow$ *since* $\mathit{next}(\sigma_l, \sigma_{l+1})$ *is the sequence of role assertions starting from* σ_0 *to* σ_k*, we have* $\forall k > 0 : \mathcal{KB} \not\models (\exists \mathit{next}.\top)(\sigma_0)$
$\Rightarrow$ $\forall k > 0 : \mathcal{KB} \not\models (\exists \mathit{next}.D_1)(\sigma_0)$
$\Rightarrow$ *since* $\forall k > 0 : \mathcal{KB} \not\models D_1(\sigma_0)$ *and* $\mathit{future} \equiv \mathit{next}$*, we have* $\forall k > 0 : \mathcal{KB} \not\models (D_1 \sqcup \exists \mathit{future}.D_1)(\sigma_0)$*. However, it contradicts with* $\mathcal{KB} \models (D_1 \sqcup \exists \mathit{future}.D_1)(\sigma_0)$.

case 2: *(b is true)*
$\mathcal{KB} \not\models D_1(\sigma_k)$
$\Rightarrow$ $\mathcal{KB} \not\models (\exists \mathit{future}.D_1)(\sigma_0)$
$\Rightarrow$ *since* $k > 0$*, we have* $\mathcal{KB} \not\models D_1(\sigma_0)$
$\Rightarrow$ $\mathcal{KB} \not\models (D_1 \sqcup \exists \mathit{future}.D_1)(\sigma_0)$*. However, it contradicts with* $\mathcal{KB} \models (D_1 \sqcup \exists \mathit{future}.D_1)(\sigma_0)$.

□

Lemma 6 *Let* $\mathcal{KB}$ *be consistent and* $\mathcal{W}$ *be a set of interpretations satisfying* $\mathcal{KB}$*.*

$$(\exists k > 0 : \mathcal{KB} \supset \{\mathit{next}(\sigma_0, \sigma_1), \ldots, \mathit{next}(\sigma_{k-1}, \sigma_k)\} \wedge \mathcal{KB} \models D_2(\sigma_k) \wedge$$

$$\forall l \in \{0, \ldots, k-1\} : \mathcal{KB} \models D_1(\sigma_l)) \vee \mathcal{KB} \models D_2(\sigma_0)$$

$$\Leftrightarrow$$

$$\mathcal{KB} \cup \{D_{aux} \equiv D_2 \sqcup (D_1 \sqcap \exists \mathbf{K}\mathit{next}.D_{aux} \sqcap \exists \mathit{future}.D_2)\} \models D_{aux}(\sigma_0)$$

Proof 9

" $\Rightarrow$ " :

$(\exists k > 0 : \mathcal{KB} \supset \{\mathit{next}(\sigma_0, \sigma_1), \ldots, \mathit{next}(\sigma_{k-1}, \sigma_k)\} \wedge \mathcal{KB} \vDash D_2(\sigma_k) \wedge$
$\forall l \in \{0, \ldots, k-1\} : \mathcal{KB} \vDash D_1(\sigma_l)) \vee \mathcal{KB} \vDash D_2(\sigma_0)$

$\Rightarrow$ $(\exists k > 0, \forall \mathcal{I} \in \mathcal{W} : \bigwedge_{0 \leq l < k} ((\sigma_l^{\mathcal{I}}, \sigma_{l+1}^{\mathcal{I}}) \in \mathit{next}^{\mathcal{I}} \wedge \sigma_l^{\mathcal{I}} \in D_1^{\mathcal{I}}) \wedge \sigma_k^{\mathcal{I}} \in D_2^{\mathcal{I}})$
$\vee \forall \mathcal{J} \in \mathcal{W} : \sigma_0^{\mathcal{J}} \in D_2^{\mathcal{J}}$

$\Rightarrow$ *Additionally, we know that*
$\forall \mathcal{I} \in \mathcal{W}, \forall l \in \{0, \ldots, k\} : \sigma_l^{\mathcal{I}} \in (D_z \equiv D_2 \sqcup \exists \mathbf{K}\mathit{next}.D_z)^{\mathcal{I}}$
$\wedge \sigma_0^{\mathcal{I}} \in (\exists \mathit{future}.D_2)^{\mathcal{I}}$ *since* $\mathcal{KB} \vDash D_2(\sigma_k)$ *and*
$\mathcal{KB} \supset \{\mathit{next}(\sigma_0, \sigma_1), \ldots, \mathit{next}(\sigma_{k-1}, \sigma_k)\}$. *Thus we have*
$\mathcal{KB} \cup \{D_z \equiv D_2 \sqcup \exists \mathbf{K}\mathit{next}.D_z\} \vDash D_z(\sigma_0) \wedge \mathcal{KB} \vDash (D_2 \sqcup D_1)(\sigma_0)$
$\wedge \mathcal{KB} \vDash (D_2 \sqcup \exists \mathit{future}.D_2)(\sigma_0)$

$\Rightarrow$ $\mathcal{KB} \cup \{D_{aux} \equiv (D_2 \sqcup D_1) \sqcap (D_2 \sqcup \exists \mathbf{K}\mathit{next}.D_{aux}) \sqcap$
$(D_2 \sqcup \exists \mathit{future}.D_2)\} \vDash D_{aux}(\sigma_0)$

$\Rightarrow$ $\mathcal{KB} \cup \{D_{aux} \equiv D_2 \sqcup (D_1 \sqcap \exists \mathbf{K}\mathit{next}.D_{aux} \sqcap \exists \mathit{future}.D_2)\} \vDash D_{aux}(\sigma_0)$.

" $\Leftarrow$ " :

$\mathcal{KB} \cup \{D_{aux} \equiv D_2 \sqcup (D_1 \sqcap \exists \mathbf{K}\mathit{next}.D_{aux} \sqcap \exists \mathit{future}.D_2)\} \vDash D_{aux}(\sigma_0)$

case 1:

$\Rightarrow$ $\mathcal{KB} \vDash D_2(\sigma_0)$.

case 2:

$\Rightarrow$ $\mathcal{KB} \cup \{D_{aux} \equiv D_1 \sqcap \exists \mathbf{K}\mathit{next}.D_{aux} \sqcap \exists \mathit{future}.D_2\} \vDash D_{aux}(\sigma_0)$

$\Rightarrow$ $\exists k > 0, \forall \mathcal{I} \in \mathcal{W} : \bigwedge_{0 \leq l < k} ((\sigma_l^{\mathcal{I}}, \sigma_{l+1}^{\mathcal{I}}) \in \mathit{next}^{\mathcal{I}} \wedge \sigma_l^{\mathcal{I}} \in D_1^{\mathcal{I}}) \wedge \sigma_k^{\mathcal{I}} \in D_2^{\mathcal{I}}$

$\Rightarrow$ $\exists k > 0, \forall l \in \{0, \ldots, k-1\} : \mathcal{KB} \supset \{\mathit{next}(\sigma_l, \sigma_{l+1})\} \wedge \mathcal{KB} \vDash D_1(\sigma_l)$
$\wedge \mathcal{KB} \vDash D_2(\sigma_k)$.

$\square$

Proof to A.1 :

$$
\begin{array}{lll}
s \vDash p_i & \Leftrightarrow & p_i \in L(s) \\
& \Leftrightarrow & \text{since } t_1 \text{ and } t_3 \text{ are injective functions,} \\
& & s \text{ and } p_i \text{ are uniquely} \\
& & \text{mapped to } x \text{ and } V_i, \text{ respectively. Obviously,} \\
& & t_8(s) \text{ is a set of tupels} \\
& & \text{that denotes the labelings of state } s \text{ in} \\
& & \text{the knowledge base representation. It means that} \\
& & (x, V_i) \in t_8(s). \\
& \Leftrightarrow & \mathcal{KB} \supset \{V_i(x)\} \\
& \overset{\text{Lemma 1}}{\Leftrightarrow} & \mathcal{KB} \vDash \mathbf{K}V_i\ (x)
\end{array}
$$

Proof to A.2 :

$$
\begin{array}{lll}
s \vDash p_i & \Leftrightarrow & p_i \notin L(s) \\
& \Leftrightarrow & \text{According to the mapping functions } t_1 \text{ and } t_3, \\
& & x \text{ and } V_i \text{ uniquely represent the state } s \text{ and} \\
& & \text{label } V_i, \text{ respectively. Thus, } (x, V_i) \notin t_8(s) \\
& \Leftrightarrow & \mathcal{KB} \cap \{V_i(x)\} = \emptyset \\
& & \text{a member of concept } V_i \\
& \overset{\text{Lemma 2}}{\Leftrightarrow} & \mathcal{KB} \vDash \neg\mathbf{K}V_i\ (x)
\end{array}
$$

Proof to A.3 :

$$
\begin{array}{lll}
s \vDash \mathbf{E}\ (g_1) & \Leftrightarrow & \text{there is a path } \pi \text{ starting from } s, \text{ such that } \pi \vDash g_1 \\
& \Leftrightarrow & \text{since } t_1 \text{ maps each state in the path } \pi \text{ into a distinct individual} \\
& & \text{and } t_9 \text{ maps each path formula } g_1 \text{ of the restricted CTL}^* \\
& & \text{into a concept } D_1, \text{ there is a sequence } \sigma \\
& & \text{starting from } x, \text{ such that } x \text{ is a member of concept } D_1. \\
& \Leftrightarrow & \mathcal{KB} \vDash D_1(x)
\end{array}
$$

Proof to A.4 :

$$\begin{aligned}
\pi \vDash g \quad &\Leftrightarrow \quad \pi_0 \vDash E(g_1) \\
&\Leftrightarrow \quad \text{the first state } \pi_0 \text{ of path } \pi \text{ has the label } p_i \\
&\Leftrightarrow \quad \text{due to the functions } t_1, t_2 \text{ and } t_9, \\
&\qquad \text{we have a sequence } \sigma, \text{ which is connected by the assertions} \\
&\qquad \mathit{next}(\sigma_i, \sigma_{i+1}). \text{ The first instance of the sequence, } \sigma_0, \\
&\qquad \text{is a member of concept } D_1 \\
&\Leftrightarrow \quad \mathcal{KB} \vDash D_1(\sigma_0) \\
&\Leftrightarrow \quad \mathcal{KB} \vDash D_1(\sigma_0)
\end{aligned}$$

Proof to A.5 :

$$\begin{aligned}
\pi \vDash g_1 \vee g_2 \quad &\Leftrightarrow \quad \pi \vDash g_1 \vee \pi \vDash g_2 \\
&\Leftrightarrow \quad \mathcal{KB} \vDash D_1(\sigma_0) \vee \mathcal{KB} \vDash D_2(\sigma_0) \\
&\Leftrightarrow \quad \mathcal{KB} \vDash (D_1 \sqcup D_2)(\sigma_0)
\end{aligned}$$

Proof to A.6 :

$$\begin{aligned}
\pi \vDash g_1 \wedge g_2 \quad &\Leftrightarrow \quad \pi \vDash g_1 \wedge \pi \vDash g_2 \\
&\Leftrightarrow \quad \mathcal{KB} \vDash D_1(\sigma_0) \wedge \mathcal{KB} \vDash D_2(\sigma_0) \\
&\Leftrightarrow \quad \mathcal{KB} \vDash (D_1 \sqcap D_2)(\sigma_0)
\end{aligned}$$

Proof to A.7 :

$$\begin{aligned}
\pi \vDash \mathbf{X}\ g_1 \quad &\Leftrightarrow \quad \pi_1 \vDash g_1 \\
&\Leftrightarrow \quad RL \supset \{(\sigma_0, \sigma_1)\} \wedge \pi_1 \vDash g_1 \\
&\Leftrightarrow \quad \text{due to the mapping functions } t_1, t_2 \text{ and } t_9, \text{ we have} \\
&\qquad \mathcal{KB} \supset \{\mathit{next}(\sigma_0, \sigma_1)\} \wedge \mathcal{KB} \vDash D_1(\sigma_0) \\
&\overset{\text{Lemma 3}}{\Leftrightarrow} \quad \mathcal{KB} \vDash (\exists \mathit{next}.D_1)(\sigma_0)
\end{aligned}$$

Proof to A.8 :
Note that the additional concept definition D_{aux} is required to define a cyclic

concept that represents the infinite sequence of σ. This additional concept is added to existing TBox to perform the reasoning.

$$
\begin{aligned}
\pi \vDash \mathbf{G}\ g_1 \quad & \Leftrightarrow \quad \forall i \in \mathbb{N}_0 : \pi_i \vDash g_1 \\
& \Leftrightarrow \quad \text{since the model only has a finite set of states, path } \pi \\
& \qquad \text{contains repeated sequences of states. Thus} \\
& \qquad \exists k > 0, \exists m \leq k : RL \supset \{(\pi_0, \pi_1), \ldots, (\pi_{k-1}, \pi_k), (\pi_k, \pi_m)\} \wedge \\
& \qquad \forall l \leq k : \pi_l \vDash g_1 \\
& \Leftrightarrow \quad \text{due to the mapping functions } t_1, t_2 \text{ and } t_9, \text{ we have} \\
& \qquad \exists k > 0, \exists m \leq k, \forall l \in \{0, \ldots, k\} : \mathcal{KB} \vDash D_1(\sigma_l) \wedge \\
& \qquad \mathcal{KB} \supset \{\mathit{next}(\sigma_0, \sigma_1), \ldots, \mathit{next}(\sigma_{k-1}, \sigma_k), \mathit{next}(\sigma_k, \sigma_m)\} \\
& \overset{\text{Lemma 4}}{\Leftrightarrow} \quad \mathcal{KB} \cup \{D_{aux} \equiv D_1 \sqcap \exists \mathbf{K}\mathit{next}.D_{aux}\} \vDash D_{aux}(\sigma_0).
\end{aligned}
$$

Proof to A.9 :

$$
\begin{aligned}
\pi \vDash \mathbf{F} g_1 \quad & \Leftrightarrow \quad \exists i \in \mathbb{N} : \pi_i \vDash g_1 \wedge \forall k \in \mathbb{N} : k < i \rightarrow RL \supset \{\pi_k, \pi_{k+1}\} \\
& \Leftrightarrow \quad \text{due to the mapping functions } t_1, t_2 \text{ and } t_9, \text{ we have} \\
& \qquad \exists i \in \mathbb{N}, \mathcal{KB} \vDash D_1(\sigma_i) \wedge \\
& \qquad \forall k \in \mathbb{N} : k < i \rightarrow \mathcal{KB} \supset \{\mathit{next}(\sigma_i, \sigma_{i+1})\} \\
& \overset{\text{Lemma 5}}{\Leftrightarrow} \quad \mathcal{KB} \vDash (D_1 \sqcup \exists \mathit{future}.D_1)\ (\sigma_0).
\end{aligned}
$$

Proof to A.10 :

This proof is almost similar to the Proof A.8. It means that we also need an additional concept called D_{aux}.

$$
\begin{aligned}
\pi \vDash g_1 \,\mathcal{U}\, g_2 \quad & \Leftrightarrow \quad \exists j \in \mathbb{N} : \pi_j \vDash g_2 \wedge \forall i \in \{0,\ldots,j-1\} : \pi_i \vDash g_1 \\
& \Leftrightarrow \quad (\pi_0 \vDash g_2 \vee \exists k > 0 : RL \supset \{(\pi_0,\pi_1),\ldots,(\pi_{k-1},\pi_k)\} \wedge \\
& \qquad \pi_k \vDash g_2 \wedge \forall l \in \{0,\ldots,k-1\} : \pi_l \vDash g_1 \\
& \Leftrightarrow \quad \text{due to the mapping functions } t_1, t_2 \text{ and } t_9, \text{ we have} \\
& \qquad (\exists k > 0 : \\
& \qquad \mathcal{KB} \supset \{\mathit{next}(\sigma_0,\sigma_1),\ldots,\mathit{next}(\sigma_{k-1},\sigma_k)\} \\
& \qquad \wedge \mathcal{KB} \vDash D_2(\sigma_k) \wedge \\
& \qquad \forall l \in \{0,\ldots,k-1\} : \mathcal{KB} \vDash D_1(\sigma_l)) \vee \mathcal{KB} \vDash D_2(\sigma_0) \\
& \overset{\text{Lemma 6}}{\Leftrightarrow} \quad \mathcal{KB} \cup \\
& \qquad \{D_{aux} \equiv D_2 \sqcup (D_1 \sqcap \exists \mathbf{K}\mathit{next}.D_{aux} \sqcap \exists \mathit{future}.D_2)\} \vDash \\
& \qquad D_{aux}(\sigma_0)
\end{aligned}
$$

Lebenslauf

Name:	Taufiq Gozali Rochaeli

Ausbildung

Januar 2004 - März 2009	Bearbeitung der vorliegenden Doktorarbeit unter der Leitung von Prof. Dr. Claudia Eckert am Fachgebiet Sicherheit in der Informationstechnik, Technische Universität Darmstadt.
Oktober 1997 - Mai 2003	Studium der Informatik an der Universität Karlsruhe, Karlsruhe; Diplom im Studienfach Informatik.
Oktober 1996 - Juli 1997	Niedersächsiches Studienkolleg, Hannover; Feststellungsprüfung.
Juni 1993 - Juli 1996	Oberschule SMAN 70, Jakarta, Indonesien.
Juli 1990 - Juni 1993	Grundschule SMPN 11, Jakarta, Indonesien.
Juli 1984 - Juni 1990	Grundschule SD Triguna, Jakarta, Indonesien.